Reachability Analysis of Fault-Tolerant Protocols

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

(Dr. rer. nat.)

durch den Fachbereich Wirtschaftswissenschaften der

Universität Duisburg-Essen

Campus Essen

Vorgelegt von

Dipl.-Inform. Sabine Böhm
aus Bochum
Essen 2007

Tag der mündlichen Prüfung:	27.04.2007
Erstgutachter:	Prof. Dr. Klaus Echtle
Zweitgutachter:	Prof. Dr. Bruno Müller-Clostermann

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftliche Mitarbeiterin der Forschungsgruppe *Verlässlichkeit von Rechensystemen* der Universität Duisburg-Essen am Campus Essen. Mein Dank gilt allen ohne die diese Arbeit nicht denkbar gewesen wäre für ihre ganz persönliche Art und Weise der Unterstützung:

Alex, André, Andre, Andreas, Anna, Anne, Arzu, Bärbel, Bruno, Carsten, Cathrin, Charlotte, Christian, Christina, Christoph, Christoph, Claudia, Conny, Daniela, Dominik, Dominik, Elisabeth, Eva, Frank, Frank, Frau Becker, Frau Beckmann, Frau Düsenberg, Frau Götze, Freimut, Gottfried, Gregor, Gudrun, Hülya, Hans, Heike, Heike, Helmut, Henrik, Herr Griebsch, Ingo, Irene, Jan, Jana, Jens, Johannes, Johannes, Jutta, Katja, Klaus, Kribbel, Leoni, Lili, Magdalena, Manuela, Manuela, Marc, Mario, Markus, Markus, Mohamed, Rainer, Ralf, Rose, Sascha, Sebastian, Silke, Simone, Sonja, Stefan, Stefan, Steffi, Sven, Tanja, Tanja, Tanja, Tanja, Tara, Thorsten, Tobias, Uli, Ulrike, Ulrike, Veronika, Werner, Wolfgang, Wolfgang.

Explizit bedanken möchte ich mich bei Prof. Dr. Echtle für spannende Diskussionen und seine Unterstützung in allen Phasen dieser Arbeit. Weiterhin danke ich Prof. Dr. Müller-Clostermann für seine konstruktiven Beiträge und seine Bereitschaft das Zweitgutachten zu erstellen.

Kurzfassung

Durch die zunehmenden Anforderungen an fehlertolerante Protokolle steigt auch deren Komplexität zusehends. Dadurch ist es deutlich schwieriger die Funktionalität der Fehlertoleranzmechanismen zu überprüfen. In dieser Arbeit wird ein modellbasierter Ansatz vorgestellt, dessen Ziel es ist "Lücken" in den Fehlertoleranzeigenschaften effizient zu finden. Dazu wird ein Algorithmus entwickelt, der eine partiellen Ordnung erzeugt und es somit erlaubt den Zustandsraum zu verkleinern ohne Verhalten bezüglich der zu prüfenden Eigenschaften zu verlieren. Weiterhin werden zwei Algorithmen zur (partiellen) Analyse entworfen, implementiert und bewertet: Der H-RAFT Algorithmus basiert auf den SDL-Elementen der jeweiligen Transitionen und erfordert keinerlei weiteres Domänen-Wissen des Benutzers. Der Close-to-Failure Algorithmus hingegen ist nur von Benutzerinformationen abhängig. Kombinationen der beiden Ansätze werden ebenfalls untersucht. Für alle vorgestellten Methoden und Algorithmen wird ausgenutzt, dass es sich um fehlertolerante Protokolle handelt. Um die neuen Ansätze mit weitverbreiteten Algorithmen vergleichen zu können wird ein Werkzeug entwickelt, welches eine einfache Integration von Algorithmen ermöglicht. Die vorgestellten Techniken werden ausführlich in Experimenten mit einem Gesamtaufwand von etlichen CPU-Monaten untersucht. Die Ergebnisse dieser Experimentreihen zeigen eindeutig die Vorteile der entwickelten Algorithmen und Methoden.

Abstract

Due to the increasing requirements imposed on fault-tolerant protocols, their complexity is steadily growing. Thus verification of the functionality of the fault-tolerance mechanisms is also more difficult to accomplish. In this thesis a model-based approach towards efficiently finding "loopholes" in the fault-tolerance properties of large protocols is provided. The contributions comprise thinning out the state space without missing behavior with respect to the validation goal through a partial ordering strategy based on single fault regions. Two algorithms for (partial) analysis are designed, implemented and evaluated: the H-RAFT algorithm is based on SDL elements constituting each transition and requires no user-knowledge. The Close-to-Failure algorithm on the other hand is purely based on user-provided information. Combination of the two algorithms is also investigated. All contributions exploit the fault-tolerant nature of the protocols. In order to compare the performances of the novel techniques to well-known algorithms, a tool has been developed to allow for easy integration of different algorithms. All contributions are thoroughly investigated through experiments summing up to several CPU-month. The results show unambiguously the advantages of the developed methods and algorithms.

Contents

Part I.

Introduction

1. Introduction

The first chapter gives an introduction into this thesis. It motivates the work, describes its goals and presents an overview of the contributions in section 1.1. The scientific background and related work follow in section 1.2. The chapter closes with the outline of the thesis in section 1.3.

1.1. Motivation and Goal of the Thesis

Design flaws of fault-tolerance mechanisms may lead to undesired consequences − in particular fault cases under very special operating conditions. Such rare "fault tolerance holes" may be very difficult to reveal. The contributions provided in this thesis aim at finding violations of fault-tolerance properties in an efficient way. Novel approaches directing the analysis towards potential weaknesses in fault-tolerance mechanisms are introduced. These validation mechanisms are based on model checking techniques, thus they operate on models of fault-tolerant communication protocols.

A particularly effective and well understood technique for validating systems of extended finite state machines is reachability analysis - variants have already been in use for almost thirty years [Zaf77]. During an exhaustive (or complete) reachability analysis, a modeled system is forced into all states that are reachable from an initial state via a sequence of execution steps. A priori defined criteria may be checked for each reachable state. These criteria may be very general, like absence of deadlocks, or highly model dependent as reception of a certain signal within a predefined time interval. By checking the criteria for every global state of the state space generated during reachability analysis, protocol properties can be validated.

Unfortunately, the state space arising from practical problems is often intractably large. For the resulting huge reachability graph exhaustive exploration is not generally feasible [CAB+98, Kur97, KG99].

The problem is caused by poor design conventions to some extent, but mainly by the unavoidable combinatorial explosion in complex systems [AALC92]. The size of the state space may grow exponentially with respect to the size of a system configuration, especially in asynchronous non-deterministic systems. Furthermore, the modeled systems and protocols are not merely becoming more complex, but they tend to represent non-terminating systems. In other words, finite state machines are replaced by infinite ones [CAB+98, ACG96] where actions or sequences of actions may be repeated an arbitrary number of times. Non-terminating systems lead to an additional growth of state space [Blo01]. Another factor adding to the complexity of the reachability graph is the inclusion of faulty components in the model. In the most universal fault model, *any component output at any time* (short: *any output at any time*), the faulty component may exhibit arbitrary behavior: it may send any value/signal to any adjacent component at any time,

even repeatedly [Böh05], increasing the complexity in both the value and time domain. Thus, it is inevitable to take measures against the enormous growth of the state space.

In this thesis contributions are made towards handling huge state spaces arising from models of fault-tolerant communication protocols. The novel techniques take advantage of the knowledge that the models under consideration should mainly be checked for design flaws in their fault-tolerance mechanisms.

In order to cope with the state space explosion problem, different strategies can be pursued. The commonly proposed strategies are partial order reduction techniques as in [GSTH96] and partial analysis techniques [Rau90] etc.

Both, the algorithms and the partial ordering technique introduced in this thesis, are based on the knowledge that models of fault-tolerance protocols are investigated. Therefore, common properties typical to fault-tolerance mechanisms can be exploited.

Partial ordering techniques have been extensively studied and many algorithms for all kinds of application areas have been proposed. Generally, those techniques exploit the independence of actions to reduce the state space of a system while preserving properties of interest. The resulting state space is equivalent to the original one with respect to the system specification [BBG04, LLEL01].

A major contribution of this thesis is the SFR-PO (single fault region partial ordering) technique to reduce the state space significantly without loss of "interesting behavior" [BE04]. The SFR-PO technique is based on single fault regions [Kes02, BE04] and explicitely tailored for use with validation of fault-tolerance protocols. In contrast to general partial ordering algorithms, actions in different single fault regions can be considered independent. Thus, the number of independent actions can be increased considerably, leading to a remarkable reduction of the state space.

While partial ordering techniques do not lead to loss of information, they may not be sufficient to reduce the state space such that an exhaustive analysis is feasible.

Partial analysis strategies try to optimize the search of the state space within given limitations of available memory and run-time. They are based on the premise that in most cases of practical interest the maximum number of states that can be analyzed is only a fraction of the total number of reachable states R. The objectives of a partial analysis are [Hol90]:

- to select i states from the complete set of reachable states in such a way that all major protocol functions are tested and/or
- to select the i states in such a way that the probability of finding any given property violation is better than the coverage i/R.

In other words, the results should be better than with a random walk through the state space. However, the ratio i/R does not take the structure of the reachability graph into account and thus is only a weak objective. The structure of the graph may be important as "interesting" behavior could be hidden in parts of the graph that can only be reached by a limited number of

paths. Furthermore, violations may be defined over several states or paths of states. Thus, the general objectives as formulated in [Hol90] will be refined below.

As another major contribution, two novel heuristic algorithms for partial reachability analysis are provided in this thesis: H-RAFT (*H*euristic *R*eachability *A*nalysis for *F*ault-*T*olerant Systems) and C2F (*C*lose *To* *F*ailure). The employed heuristics are focused on increasing the chances of exploring those parts of the state space leading to violations of the fault-tolerance properties claimed for the protocol. Thus, the algorithms concentrate on the second objective for partial analysis.

Due to the incomplete nature of partial analysis, not all fault-tolerance violations may be found [Laf03, Blo01]. The purpose of the contributed algorithms is thus to increase the chances of finding violations with respect to existing algorithms.

Chances are increased by applying heuristics directing the search through educated guesses. The novel algorithms are considered successful if

1. they find fault-tolerance violations that have not been detected by general algorithms for reachability analysis and/or (in case of a partial analysis)
2. they find fault-tolerance violations faster than the general algorithms. In other words: less transitions had to be performed.

The algorithms are designed to work on PCs with todays computation speed and memory limits. Furthermore, it is assumed that run-time is limited. In industrial practice, results are expected quickly, especially during the development phase of a protocol. If checking model properties takes too long during protocol development this would not be acceptable.

The basis of both algorithms is to determine weights for the transitions based on their expected probability to be on a path leading to a fault-tolerance violation. Selection of the parts of the state space to be explored more thoroughly is then based on these weights.

The H-RAFT algorithm is based on the language elements of communicating automata. As representative modeling language, the *Specification and Description Language* SDL has been chosen. Weights are calculated according to the elements constituting each transition. For each element type a static weight is determined expressing the importance of the element type with respect to fault tolerance. The advantage of this approach is fast off-line weight determination. Furthermore, it requires the user to supply only a minimum knowledge about the actual model. So, the algorithm can be applied to all models representing fault-tolerance protocols.

The second heuristic, resulting in the C2F algorithm, is highly based on user information about the model. The user may specify events and conditions that are likely to represent faulty behavior eventually leading to a fault-tolerance violation. Different events and conditions may be combined in general rules. In this algorithm, these user-defined weights form the basis for calculation of the transition weights. The performance of this algorithm is highly dependent on the ability of the user to specify valuable information. Section 7 is dedicated to this algorithm.

The H-RAFT and the C2F algorithms can also be merged, thus combining fast static weight calculation with valuable user information. The weights of the resulting algorithm are combinations of the weights of the two algorithms.

Further techniques for reduction of the state space independent the fault-tolerant nature of a protocol are also contributed. Those reductions are based on distinguishing the initialization and analysis parts of the model from the main parts implementing the protocol.

Despite many available commercial and academic tools for reachability analysis, the novel techniques and algorithms have been implemented in a new tool: RAFT. Most commercial tools lack the ability to include new algorithms and other reduction techniques. Academic tools often require special modeling languages or are designed for special purposes other than detecting fault-tolerance violations. Thus, including the novel techniques is not feasible in most cases. RAFT is designed to allow for easy addition of further algorithms. It also comprises, among others, an SDL-to-Java compiler, a graphical user interface and means for analysis of message sequence charts (MSC).

In summary, the main contributions of this thesis towards efficient reachability analysis aiming at finding design flaws in fault-tolerant communication protocols are:

- **SFR-PO**: A partial order reduction strategy based on the definition of single fault regions;
- **H-RAFT**: A heuristic reachability analysis algorithm refraining from additional user input;
- **C2F**: A heuristic reachability analysis algorithm exploiting user-provided information;
- **Extensive Analysis** of all contributions and comparisons to existing algorithms.
- **RAFT**: A tool providing
  - an implementation of all contributions,
  - an SDL-to-Java compiler,
  - support for specifying queries,
  - support for MSC analysis,
  - an interface for easy integration of additional algorithms;

The first four items represents the scientific contributions of the thesis. The purpose of the RAFT tool is merely to provide a comfortable environment for experimental evaluation.

1.2. Scientific Background and Related Work

Analyzing models of protocols has several advantages over formal verification techniques or applying test-cases. Achieving adequate coverage by formal verification methods is hardly feasible for large protocols [Pol95]. Similarly, generating and applying a sufficient amount of test-cases is much too time consuming [TCL99, GF93, PB03]. Although, compared to tests of the real system, the model-based approach is less detailed due to the inherent abstractions [BGPQ02]. The need for a fault injector [BT97, DJMT96, EL95] is eliminated, and thus the problem of selecting representative faults for injection is not given. First steps towards approaches of combining model checking and (incremental) generation of hardware test-cases are given in [ABCS01, KW91].

Another major advantage of model checking [CGP99, YTK01, BFG02] is its applicability at a very early stage of protocol development [ACG96]. Most of the faults the designer is thinking of can be included into the model [Joc02] even before the real system is implemented. Thereby, the results of testing can be included into the development process before prototypes are required.

1.2.1. Related Tools

When modeling a protocol, abstractions have to be made. These abstractions do not necessarily present a disadvantage of model-based analysis. Different abstraction levels may be used depending on the parts of a protocol one is interested in. Parts that are not of current interest, or have been shown to work correctly before (for example CRC calculations), can be modeled very coarsely, while protocol parts of high interest may be modeled very fine grained. This idea has been adopted by Cobleigh et.al. [CCO02]. They present a tool capable of creating imprecise, coarse grained models from Ada or Java programs. The user is then assisted by adding details as needed.

Furthermore, protocols are usually designed in a hierarchical structure to cope with their complexity. In this design, some levels within the hierarchy may be considered abstractions of other levels. Additionally levels may be specified in different abstraction levels – especially with respect to the fault-tolerant behavior. Thus, modeling can exploit the already existing level structure.

The advantages of model checking are also valued in the industry - verification through model validation is of increasing interest. For example, in [SRSP04], the (large) TTP/C protocol [TTT03, KG93] has been investigated by applying model checking techniques. Many case studies of well known protocols of all kinds of areas have proven the adequacy of model checking techniques for protocol validation. Examples of these studies from the automotive, aerospace, real-time multimedia and many other areas can be found in [CAB+98, JPP+97, Sev93, TCL99, BGK+96, CGP02, TAML00]. Integration efforts to introduce the achievements made in the academic community into industrial practice are also a current research topic [LH04, LH02, CT97].

The different model checking techniques are based on a variety of modeling languages and model types depending on the focus of the analysis. They range from timed automata [Bro91] to petri nets [BK02], from SDL [ITU93b] to academic languages defined for a single tool only, [Hol97] etc.

SDL [ITU93b] and other formal description techniques, such as LOTOS [ISO88], and their related formalisms such as MSC [ITU93a] and TTCN [ITU01] are highly suitable for specification and validation tasks of telecommunication systems [BFG+99]. They are also very popular because commercial tools, like SDT [Tel01], supporting those description languages are available. The tools mainly provide support for requirement analysis, graphical editing means, code generation and testing. The wide-spread use of these techniques [LH00] in the community of telecommunication systems is also due to the standardization efforts of the ITU and other international standardization bodies. SDL has been standardized by the ITU (formerly CCITT) in [ITU93b] and the formal semantics of the language are defined in [ITU94b, ITU94a].

Since SDL is based on extended state machines [Bro91] communicating asynchronously via queues, it is very well suited for modeling component based systems like communication protocols

[HP89] in general [CW00, Sev93]. SDL is also of increasing interest to hardware specifiers [TCL99] as it offers rigorous specification possibilities as well as intuitive system structuring features. Furthermore, it allows for high-level communication specification and contains means for hardware-software co-design. In these respects, it complements industrial hardware languages such as VHDL (*V*ery High Speed Integrated Circuit *H*ardware *D*escription *L*anguage [IEE93]) and VERILOG [IEE95].

The main goal of modeling hardware in SDL is synthesis with the respective software. For this purpose, standard engineering tools are used [TAML00]. SDL hardware descriptions are often translated into VHDL [BF95, DMVJ93, GKRM93]. Thus, SDL can be used for high-level hardware description and can be coupled with common tools for hardware synthesis and further analysis. In [GRK93, JRV+97, LBBI96] investigations on hardware-software co-design using SDL have been presented. Available tool-sets for this purpose include COSMOS [DMIJ97] and ODE [HS96].

The academic community also puts effort in techniques and tools to ease the use of SDL for the hardware community. In [CT97], the ANISEED (*AN*alysis *I*n *S*DL *E*nhancing *E*lectronic *D*esign) is presented for comfortably modeling digital logic in SDL. Its applicability has been shown in several case studies, [TCL99, CT97] for example. Most case studies and application areas, considered in research so far, focus on relatively small resulting models. However, SDL has also been recommended for validation of large-scale industrial systems [ACH+96].

Alternatives to the Specification and Description Language have been proposed in [Bro91, God91, BMU98], for example. However, these alternative languages are not very popular as they lack support of commercial tools and are not yet accepted in the industrial community. The wide-spread use of SDL in different application areas has been the reason for choosing SDL as the modeling language in this thesis. Nevertheless, the novel techniques can also be applied to other languages describing communicating timed automata.

1.2.2. Related Scientific Work

With the growing complexity of the modeled protocols, the state space grows rapidly. This state space explosion problem is not only an academic one, but has also been observed in several real-life case studies [LH04, CAB+98]. Many tool-sets for reachability analysis contain a partial ordering mechanism.

Despite the reduction achievable through partial ordering, the state space is often still too large to be explored completely. Thus, partial analysis strategies have been developed. These strategies are based on influencing the direction of analysis more or less sophisticated by defining selection criteria for choosing the states to be analyzed further. Possible selection criteria for guiding the partial analysis are random selections, straight-forward selections as in pure depth-first and pure breadth-first search and heuristics.

The first three strategies are so-called “blind” strategies [LLEL01]. They are very general methods applicable to all systems. Heuristics, on the other hand, exploit the availability of information about the system to guide the search in a more sophisticated way. Many heuristics have been motivated and proposed for different analysis goals. Examples can be found in

[GSTH96, LLEL01, Hol87b, Hol88, DM04, Blo01, Rau90, Wal96]. The most famous heuristic, the bitstate (or supertrace) algorithm, has been introduced in [Hol87b, Hol88]. This algorithm reduces the memory requirements drastically at the cost of missing behavior. Dillinger and Manolios proposed a method to make this algorithm more reliable in terms of missing less behavior [DM04]. The bitstate algorithm is described in more detail in section 3.2.3. The approach presented in [BGPQ02] is into the same direction: Memory requirements are eased by utilizing abstractions, mainly on the variables of the modeled protocol. The authors of [ACG96] cope with the size problem by analyzing software artifacts separately. This approach has the inherent risk of missing undesired behavior at the interfaces between the artifacts.

Grabowski et.al. [GSTH96] follow a different approach. They restrict the type of systems to be analyzed to *closed* systems. In a closed system, no stimulus from outside the model is allowed, thus no interactive systems may be modeled unless a model of the environment is included.

While the algorithms presented throughout this section provide means for handling the state space explosion problem in general or in special application cases, none of them considers fault-tolerant applications explicitely. Since those applications are of increasing interest to the industrial community, development of algorithms specifically aiming at finding design flaws in the conceived fault-tolerance mechanisms is an important task. In [EN99], approaches in this direction have been proposed. The authors present algorithms for analysis of safety-critical systems. Their highly safety-specific Close-to-Danger (C2D) algorithm is based on "switches" indicating the distance to danger. The Close-to-Failure (C2F) algorithm introduced in this thesis is based on the ideas of the latter algorithm. Although the results of C2D were not overwhelming when applied to safety-techniques (compared to a coverage algorithm), the adaptation to fault-tolerant systems may yield better results.

Apart from the H-RAFT and C2F algorithms, this thesis contributes several algorithms and methods for efficiently validating fault tolerant communication protocols. Efficiency can be considerably increased by taking the fault-tolerant nature of the systems into account.

1.3. Organization of the Thesis

This thesis is structured as follows: Throughout the remainder of the first part, an introduction to SDL (chapter 2) and to reachability analysis including basic algorithms (chapter 3) is provided. These chapters provide information to readers not familiar with SDL respectively reachability analysis. Furthermore, they represent a reference for higher level descriptions in subsequent chapters.

Part II contains a detailed description of the contributions of this thesis: Chapter 4 is dedicated to the state space reduction methods, focusing on the SFR-PO technique. Chapters 6 and 7 provide the H-RAFT and C2F algorithms, respectively.

In part IV, the techniques and algorithms presented in part II are analyzed and evaluated. Different parameterizations and combinations of the techniques are compared to each other and to existing algorithms. For this purpose, several models of fault-tolerant communication protocols have been implemented to substantiate any improvements. Descriptions of the implemented protocols are also provided.

The RAFT tool is summarized in part III. The focus of the description is on providing information required for utilizing the tool.

Part V contains a summary of the contributions and their evaluation, followed by an outlook at future research directions.

2. SDL

Development of the *Specification and Description Language SDL* started in 1974. In 1988 it has been standardized by the CCITT (now ITU). Several updates of the standard have been defined by the ITU. The following description of SDL focuses on SDL-92 [ITU93b].

2.1. Introduction

The *Specification and Description Language SDL* [ITU93b] is based on communicating extended state machines (cESM). It is specified in two notations. The textual representation denoted by SDL/PR (*PR*intable) and the graphical representation SDL/GR. Both notations are equivalent w.r.t. their expressiveness and may be converted into each other. This section gives an introduction to both representations. Throughout the thesis, the graphical representation will be mainly used for illustrations. The textual representation is the basis for the novel algorithms introduced in part II.

The description of SDL provided in this section does not cover all aspects of the language. It is restricted to those concepts required for further understanding. For a complete definition the reader is referred to [ITU93b]. The introduction given here is geared at giving a coarse overview of the structure of the language. A more detailed description of elements is given, if this knowledge is required, in later parts of the thesis.

2.2. Hierarchy

SDL is based on a hierarchical structure consisting of `system`, `block` and `process` levels, as shown in figure 2.1. The highest level is the `system` level. A system may contain several `block`s, representing the next lower level. At least one block is required in an SDL system. Blocks in turn contain `processes`. Processes are located at the lowest level. Each process represents a local ESM with states and transitions defining its program flow. Processes located within the same block communicate via `signalroutes`. Communication between blocks is established through `channels`. Blocks are mainly used for structuring purposes. Basically, there is no functional difference whether processes are located in the same block or in different blocks. The only difference is that channels may impose delays while signalroutes don't (see also paragraphs "Channels", page 13, and "Signalroutes", page 14)

System Level. Figure 2.2 shows the SDL system level of an example system *commEx*. The graphical representation (SDL/GR) is depicted in figure 2.2(a). Figure 2.2(b) is the textual representation (SDL/PR) of the same system.

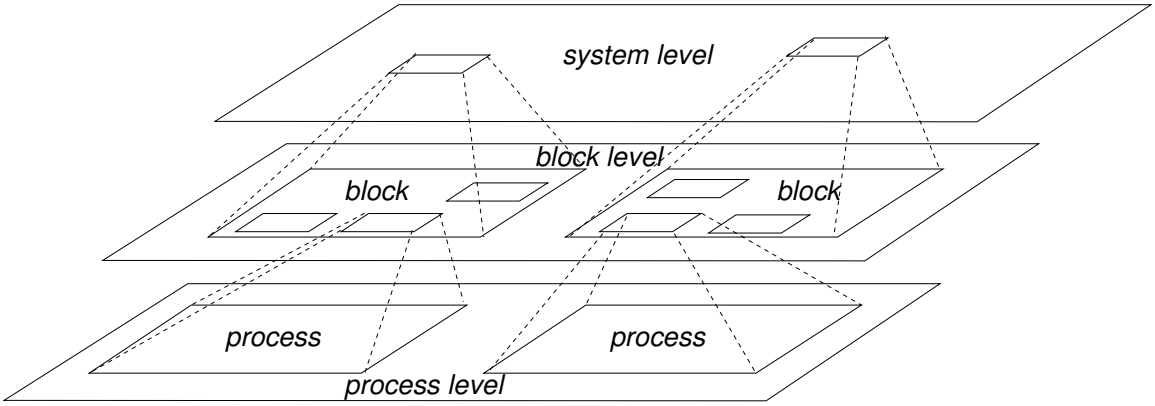
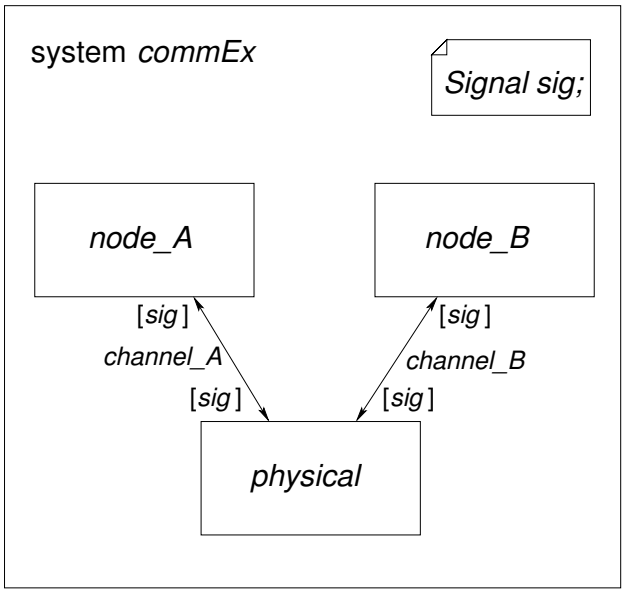


Figure 2.1.: SDL Hierarchy.



(a) System Level (Graphical Representation).

```
system commEx;
    Signal sig;
    block node_A referenced;
    block node_B referenced;
    block physical referenced;
    channel channel_A  NODELAY
        from node_A to physical
            with sig;
        from physical to node_A
            with sig;
    endchannel;
    channel channel_B  NODELAY
        from node_B to physical
            with sig;
        from physical to node_B
            with sig;
    endchannel;
endsystem commEx;
```

(b) System Level (Textual Representation).

Figure 2.2.: SDL System Level.

On the system level the following items can be defined:

- **Block References:** In system *commEx*, three blocks are defined: *node_A*, *node_B* and *physical*. In SDL/GR, blocks are represented by rectangles, in the textual representation by `block` *blockName* `referenced`. The keyword `referenced` indicates that the respective block is defined outside the `system` ... `endsystem` part.

- **Signals:** Signals that may be sent from one block to another have to be specified on the system level. In the example, only one signal *sig* is defined. The syntax in SDL/GR and SDL/PR is similar: `Signal` *signalName*. In SDL/GR this has to be set in a text box, represented by a rectangle with folded edge.

- **Channels:** Channels are the communication routes between blocks. Their origin and destination blocks have to be defined and the signals allowed via that channel need to be named. In the example, two channels, *channel_A* and *channel_B*, are defined between block *node_A* and block *physical* and between block *node_B* and block *physical*, respectively. In SDL/GR this is represented by arcs between the blocks. In SDL/PR a channel description is placed in a `channel` *channelName* ... `endchannel` environment. The syntax for specifying the blocks that will be connected by the channel is `from` *originBlock* `to` *destinationBlock*. In the example the channels are bidirectional. Thus, the defined signals may be sent in both directions. Bidirectional channels are indicated by double-headed arcs. It is also possible to define unidirectional channels. Furthermore, multiple channels between blocks may be defined. In the example only signal *sig* is allowed via the two channels. In SDL/GR this is indicated by `[`*sig*`]`. The position indicates which block is allowed to send *sig* on the channel (here: all blocks may send *sig*). In SDL/PR allowed signals are indicated by a preceeding `with`.

  Blocks may also be connected to the environment (not shown in figure 2.2). Then a channel from a block to `env` has to be defined. In the graphical representation this is indicated by an arc leading to the outer frame of the `system` rectangle. In the textual representation *origin* and/or *destination* are set to `env`.

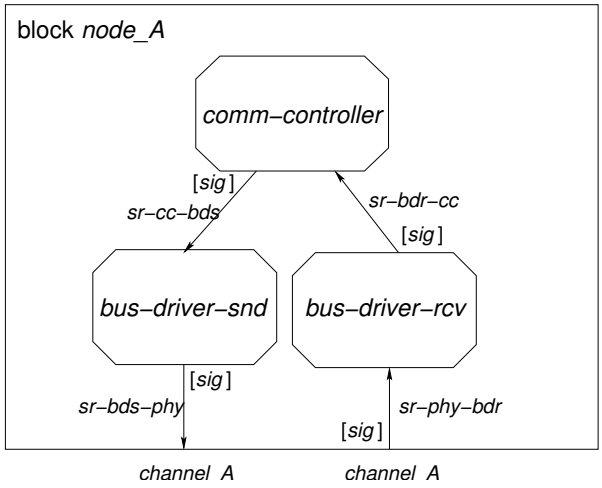
  The environment of an SDL system is defined as the surrounding that is not part of the system itself, but communicates with the SDL system. Thus, it is possible to model only parts of the complete system and test the interactions between the modeled parts and the real system.

  Each channel can be defined to impose either zero-time delay on each signal passing it, or to delay each signal for "an indeterminant and non-constant time interval" [ITU93b]. A zero-delay channel is indicated by `channel` *channelName* `NODELAY` in the textual representation and by placing the arrowheads at the end of the connecting lines in SDL/GR. In figure 2.2, channels with zero delays are defined. Channels with delay are represented by placing the arrowhead in the middle of the connecting line, respectively omitting the `NODELAY` keyword.

- **Data Type Definitions and Synonyms:** Apart from the items shown in the example, new datatypes and synonyms may be defined on the system level. Datatypes can be either specified as subtypes of an existing type by `Syntype` or as entirely new types by `Newtype`. `Synonyms` on the system level represent system-wide constants. Knowledge of the exact

syntax of data type definitions and synonyms is not required in subsequent sections, thus it will not be discussed here.

Block Level. Figure 2.3 shows block *node_A* of system *commEx* as an example of an SDL block. The following items can be specified on the block level:



(a) Block Level (Graphical Representation).

```
block node_A;
    process comm–controller referenced;
    process bus–driver–snd referenced;
    process bus–driver–rcv referenced;
    signalroute sr–cc–bds
      from comm–controller to bus–driver–snd
          with sig;
    signalroute sr–bds–phy
      from bus–driver–snd to env
          with sig;
    ...
    connect channel_A and sr–bds–phy;
    ...
endblock node_A;
```

(b) Block Level (Textual Representation).

Figure 2.3.: SDL Block Level.

- **Process References:** Block *node_A* contains 3 processes *comm-controller*, *bus-driver-snd* and *bus-driver-rcv*. They are represented as rectangles with cut edges in SDL/GR and by `process` *processName* `referenced` in SDL/PR. Again, `referenced` indicates that the processes are defined outside the `block` ... `endblock` environment.
- **Signalroutes:** Communication between the processes located in the same block is established through signalroutes. The representation of signalroutes is similar to the one of channels. In SDL/GR arcs with the name and the allowed signals of the signalroute are used. In SDL/PR the syntax differs from the channel syntax only in that no `endsignalroute` is required. In figure 2.3(b) only two of the four signalroutes are depicted, the other ones are specified accordingly. In this example, all 4 signalroutes are unidirectional. Bidirectional routes can be specified by two-headed arcs (SDL/GR) respectively by adding the other direction as for channels in SDL/PR.
- **Connections:** Signalroutes connected to the environment (the rectangle around block *node_A* respectively the `env` *origin/destination*) may be connected to channels thus establishing communication between processes in different blocks. In SDL/GR the *channelName* is specified at the respective arc outside the block rectangle. In SDL/PR the connection is expressed by `connect` *channelName* `and` *signalrouteName*.
- **Signals:** Signals that are only sent between processes in the same block may be specified on the corresponding block level instead of the system level as they need not be visible

outside the respective block. The syntax for specifying signals on the block level is the same as on the system level.

- **Data Type Definitions and Synonyms:** Data types and synonyms may also be specified on the block level instead of the system level if they are only used in the processes specified in the respective block.

Process Level. On the process level, the local state machines (the processes) are defined. Processes require a more detailed description provided in the following section (2.3).

2.3. SDL Processes

Each process specifies an automaton representing the behavior of a component that is part of the modeled system. The automaton consists of states and transitions between those states. The basic structure of a process is described in section 2.3.1. Sections 2.3.2 and 2.3.3 give a detailed description of triggering events and of actions that may be performed during a transition.

2.3.1. Process Overview

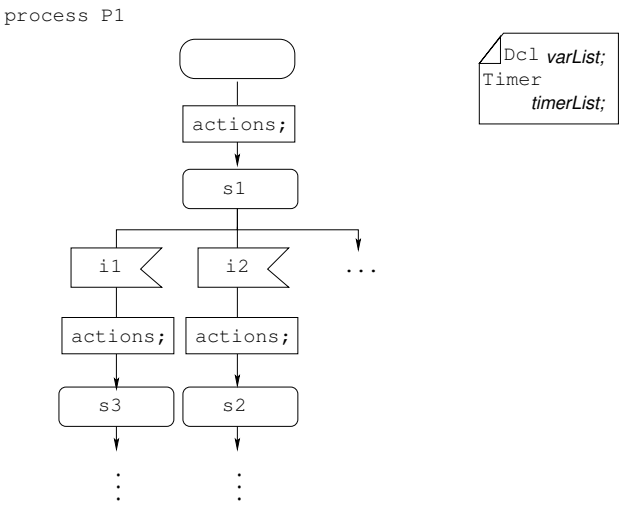
Figures 2.4(a) and 2.4(b) show the basic process structure of an illustrative example process *P1*. First, the variables and timers are declared. In the graphical notation these declarations are placed in a text box.

Variable Declaration. A variable declaration begins with the keyword `Dcl` followed by a comma-separated list, denoted by *varList* in the examples, of the declaration of each variable. The syntax for each variable declaration is *variable variableType*.

Timer Declaration. Timers are declared in a comma-separated list, *timerList* in the examples, preceeded by the keyword `Timer`. The list contains all timer names. SDL allows for specification of timer arrays, apart from "normal" timers. Timer arrays are represented by *timerName(indexType)*. The *indexType* has to be a (sub-)set of the natural numbers.

After the declarations, the states and transitions representing the program flow of the process are defined.

Start Transition. Each process begins with a `start` transition. This transition is executed upon creation of the process without any stimuli being present. The *actions* performed in this transition are usually used for initialization purposes. However, they are not limited to this purpose. Any actions defined in SDL may be specified here. A detailed list of all actions is provided in section 2.3.3.



(a) Example SDL Process (Graphical Representation).

```
process P1;
    Dcl varList;
    Timer timerList;
    start;
      actions;
      nexstate s1;
    state s1;
      input i1;
        actions;
        nexstate s3;
      input i2;
        actions;
        nexstate s2;
      ...
    endstate s1;
    state s2;
      ...
    endstate s2;
    ...
endprocess P1;
```

(b) Example SDL Process (Textual Representation).

Figure 2.4.: SDL Process Level.

Nextstate. Each transition, independent of whether it is a start transition or a "normal" transition, has to indicate its successor state. In the textual representation, `nextstate` *s1* `;`, for example, expresses that process *P1* is in state *s1* after the transition has executed. In the graphical notation the next state is indicated by an arc. Transitions back to the same state are represented by an arc originating and ending at the same process in SDL/GR. In SDL/PR, `nextstate -;` can be specified as an abbreviation instead of indicating the state name again.

Stop. The keyword `stop` indicates the termination of the process. It can be placed in the model instead of a `nextstate` expression. In the graphical representation, `stop` is depicted by a large X-shaped symbol.

States. Local states of a process are indicated as rectangles with rounded edges in SDL/GR. The name of the state is placed inside the rectangle. In the textual representation each state is encapsuled in `state` *stateName* ... `endstate` *stateName*.

Asterisk State. Special states denoted by `state *` in SDL/PR and by the state symbol with the asterisk inside instead of the state name in SDL/GR, can also be specified. The syntax within the state definition is the same as for normal states. However, the contents of the state (the inputs, actions, nextstates) are appended to each normal state specified in the process. Thus, the asterisk state is a convenient shortcut for specifying the same transitions in all of the normal states. Furthermore, readability of the model is increased.

Multiple asterisk states may be specified in a process as long as this does not yield any transition being defined multiple times in a single state. Each of the asterisk transitions may be equipped with an exclusion list – a list of states the transitions should not be appended. Exclusion lists are set in brackets after the asterisk.

In SDL/GR, asterisk states are also specified like normal states, however, there are no incoming arcs to the states labeled `*`.

Transitions. Transitions that are not start transitions are enabled by an input element. In the graphical notation, inputs are denoted in flag-shaped symbols with the input name inside. In SDL/PR they are preceeded by the keyword `input`. Input elements are subject to section 2.3.2. The remainder of each transition consists of actions (optionally) and the successor state.

2.3.2. Input Elements

Table 2.1 depicts the input elements relevant for this thesis. For each element the textual and the graphical representation in SDL is given. Reserved words are set in typewriter style. Italics indicate variables. Each input element including the save expression (which is a special input element) in the last row are described in subsequent paragraphs. Input elements are stored in an input queue of the receiving process in the order of their arrival. They remain in the queue until they are consumed. Consumption of an input element fires the associated transition.

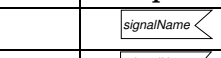
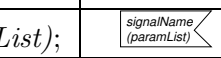
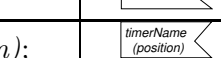
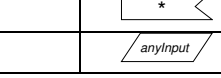
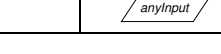
Input Element	Textual	Graphical
signal	`input` *signalName*;	
signal with parameters	`input` *signalName(paramList)*;	
timer	`input` *timerName*;	
timer array	`input` *timerName(position)*;	
spontaneous	`input none`;	
asterisk	`input *`;	
save	`save` *anyInput*;	

Table 2.1.: SDL Input Elements.

Signals. Signals are used for establishing communication among processes and between processes and the environment. The *signalName*s have to be defined on the process or block level as described in section 2.2. The input element `signal` indicates that the transition is enabled if the signal specified by *signalName* is in the input queue of its process.

Signals may also carry parameters in a parameter list (*paramList*). The variables of the *paramList* have to be declared in the receiving process (see paragraph "Variable Declaration" in section 2.3.1). The variables are set to the values transmitted in the *paramList* upon firing the transition (see also paragraph "Sending Signals" on page 20). In other words: parameters are passed "by value".

Timers. Transitions specifying a timer *timerName* as input element are enabled if the corresponding timer event is in the input queue of the process, because the timer has expired. If the timer is part of an array, the position of the timer in the array is also provided in the parameter *position*. *timerName* has to be specified in the timer declaration list of the process (see paragraph "Timer Declaration" in section 2.3.1). As for parameters of signals, the variable holding the position has to be declared in the process and is set once the transition fires.

Spontaneous Transitions. Spontaneous transitions in SDL may be activated without any input being present in the input queue of the process. There is no priority between spontaneous and "normal" transitions. In other words: A spontaneous transition defined in state s_i of a process may fire at any time while the process is in state s_i. This includes not firing the transition at all.

Asterisk. Asterisk transitions are enabled by every input that is not specified as input to any other transition in the same state. Thus, they resemble the idea of default transitions. Note, that asterisk transitions are not required in a state. If no asterisk transition and no save transition (see below) is specified within a state, signals contained in the input queue of the process may be dropped if they cannot be consumed immediately, because they are not specified in any of the current state's transitions. An exclusion list can be specified for an asterisk transition indicating the input elements, apart from the ones already specified in other inputs of the transition, the transition shall not be enabled by.

Save. If a signal/timer is in the input queue of a process, but cannot be consumed in the current local state, it is discarded and thus lost. The loss can be prevented by the `save` construct. Signals/Timers that are specified in the *anyInput* list of a state are not lost, but remain at the same position in the input queue. The signals and timers specified in the *anyInput* list may not be specified as input elements within the same state. The asterisk, with the same meaning as in `input *`, may also be specified as *anyInput*. Then, all signals that are not specified as input elements in the current state remain in the input queue.

Input Lists. Transitions differing only in their input elements may be summarized by not specifying only a single input element to that transition, but to provide a comma-separated list of input elements. Signals, timers and spontaneous elements may be specified in arbitrary oder in the input list. Obviously, the asterisk is not allowed in the list.

2.3.3. Action Elements

Once a transition fires through consumption of a signal, several actions may be performed, before the state is changed. In this section, the SDL action elements are described as far as they are relevant for this thesis. Table 2.2 gives an overview. The single action elements are discussed in subsequent paragraphs.

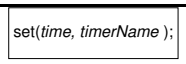
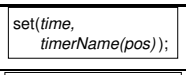
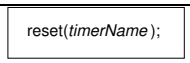
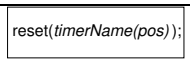
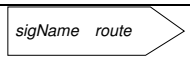
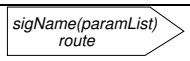
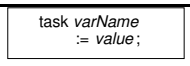
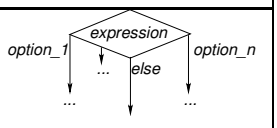
Action Element	**Textual**	**Graphical**
Setting timer	`set`(*time*,*timerName*);	
Setting timer in array	`set`(*time*,*timerName(pos)*);	
Resetting timer	`reset`(*timerName*);	
Resetting timer in array	`reset`(*timerName(pos)*);	
Sending signals	`output` *sigName route*;	
Sending signals with parameters	`output` *sigName(paramList) route*;	
Changing Variables	`task` *varName* `:=` *value*;	
Decisions	`decision` *expression*; *option_1: actions*; ... *option_n: actions*; `else:` *actions*; `enddecision;`	

Table 2.2.: SDL Action Elements.

Setting Timers. Timers may be set relative to the current model time or absolute. Setting a timer absolutely is equivalent to setting it relative to the start time zero. When setting the timer absolute, its expiration time has to be provided for the variable *time*. In order to set the timer relative to the current model time, which is denoted by the SDL keyword `now`, *time* is set to `now`+duration, where the duration must be a non-negative value. Timers in timer arrays can be set absolute or relative as well. They require the additional parameter *pos* specifying the position of the desired timer in the array.

Resetting Timers. Resetting timer *timerName* in SDL results in the timer being stopped and discarded until it is set again. Resetting timers in arrays requires providing the position *pos* within the array.

In order to change the expiration time of a timer, it is not necessary to reset and then set it again. Setting of a timer implicitly includes its reset.

Sending Signals. When sending a signal *sigName*, its destination (*route*) has to be provided. The *route* can be specified in several ways: It may indicate the receiving process directly. The syntax for *route* is then: `to` *receiving process*. For example: `output` *sig* `to` *P2*. Instead of indicating the receiving process, the outgoing signalroute can be specified for *route* by `via` *signalroute*: `output` *sig* `via` *sr-phy-bdr*. *route* may be omitted, if the signal is only specified on one signalroute originating at the process, in other words: its path is non-ambiguous. If no receiving process and no signalroute is specified although several paths exist, the signal is sent via an arbitrary route. This results in an undesired growth of the global state-space.

Changing Variables. The content of a variable *varName* can be changed to value *val* by the `task` action. Setting single fields of variables of struct-based types is possible through `task` *structName*`!`*fieldName* `:=` *value*. Setting all fields of a struct variable at the same time is denoted by `task` *structName* `:= (.` *field_1, ..., field_n* `.)`. It is also possible to assign one struct variable to another one of the same type. For example: `task` *struct_Y* `:=` *struct_X*.
varName(pos) has to be specified for altering an element of an array-based variable *varName* at position *pos*.

Decisions. Decisions represent the branching mechanism of SDL. Their basic syntax is shown in the last row of table 2.2. The *expression* can be any statement that resolves to a value of any type *decisionType*. For example, a boolean expression, a calculation or any variable value. The options *option_1* to *option_n* have to be of the same *decisionType*. The first match, where *expression* evaluates to the same value as *option_i*, results in the execution of the tasks of that option. After execution of the actions associated with *option_i*, the decision is left. No actions of later options will be executed. The *actions* of each option are specified in the same way as the actions outside a decision. A default option that is executed if no other option is applicable is indicated by `else`.

Expression may also be set to the keyword `any`. In this case, the *options* are omitted, thus each branch starts with a colon (in the textual representation). A random selection is applied to determine the branch to be executed within the global state space.

2.4. Concept of Time in SDL

In most cases, highly fault-tolerant systems are also real-time systems. Thus, it is inevitable to consider time and timing concepts when coping with fault-tolerance protocols.

In the description of setting timers in SDL (see 2.3.3), the SDL variable `now` has been introduced. `now` contains the current model time, which is specified in units. The mapping of these units to real time is not predefined, but has to be accomplished by the modeler. The units should represent the finest time resolution required in the model. It is up to the modeler to convert all timing expressions to the finest granularity. At system start `now` is set to zero.

SDL defines no standard semantics of time. Interpretations applied in tools range from considering all transitions *eager* to considering all transitions *lazy* [BFG+99]. In the first approach, time may only advance if no transition can be fired at the current model time anymore. The latter approach assumes that time can progress always. Different timing concepts are discussed in section 5.1.3.

3. Reachability Analysis

Fault tolerance (and other) properties of a protocol can be checked by performing reachability analysis on the global state space of the model. The method is wide-spread in the academic area and also gains importance in industrial projects. Many (specialized) tools and languages are available for performing efficient reachability analysis for the respective analysis goals. In this chapter, a short introduction to reachability analysis and validation is given. A classification of different exploration strategies is provided and several application areas including tools and algorithms are summarized.

3.1. Introduction to Reachability Analysis

In reachability analysis, all possible execution sequences of the concurrent automata constituting the system are generated. Starting at the initial global state, all reachable global states can be generated successively resulting in a reachability graph. If all backward-edges (with respect to the applied exploration strategy), representing reconvergences, are removed, the reachability graph is turned into a reachability tree with inner nodes and leaves. Leaves may either represent nodes preceeding a reconvergence or a deadlock.

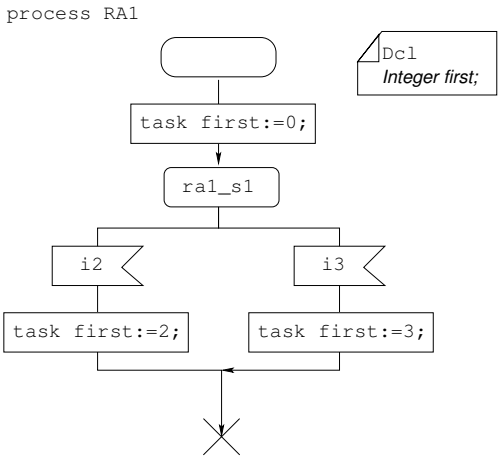
Example. Figure 3.1 shows an example SDL model of three (communicating) processes: *RA1*, *RA2* and *RA3*. *RA2* and *RA3* are almost identical: They both send a signal (`i2`, respectively `i3`) to process *RA1* at time four and then terminate. *RA1* waits for reception of the first signal arriving from either *RA2* or *RA3*. Depending on the signal, it sets variable `first` and then terminates.

Figure 3.3 depicts the reachability graph as generated by an exhaustive exploration of the SDL model.

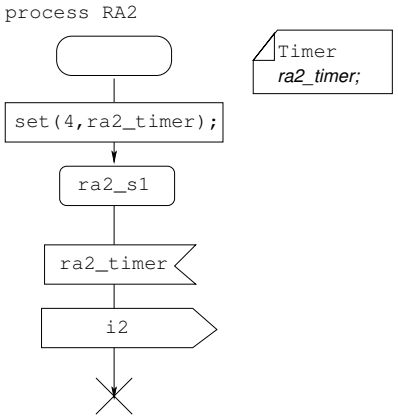
Each global state of figure 3.3 comprises the local states of the three processes and the current model time arranged as shown in figure 3.2(a). Each of the local states (see fig. 3.2(b)) contains the current local state name in the first line, followed by the current variable value for process *RA1* respectively the expiration time of the timer for processes *RA2* and *RA3*. The last line contains the current input queue in FIFO order.

In the initial global state of figure 3.3 (level 1), all processes are in their `start`-state. Neither variables nor timers are set. The input queues are all empty and system time is initialized to zero.

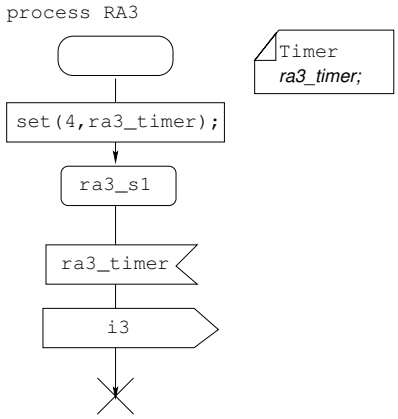
From this root state, three transitions are possible. Each process may fire its *start transition*. Levels 2 to 4 contain only the different orders of firing those start transitions.



(a) Process RA1.

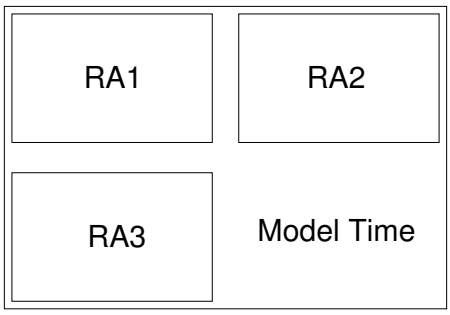


(b) Process RA2.

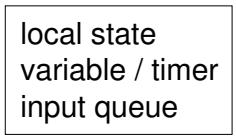


(c) Process RA3.

Figure 3.1.: Reachability Analysis Example - SDL Code.



(a) Global State.



(b) Local State.

Figure 3.2.: Reachability Analysis Example - Overview.

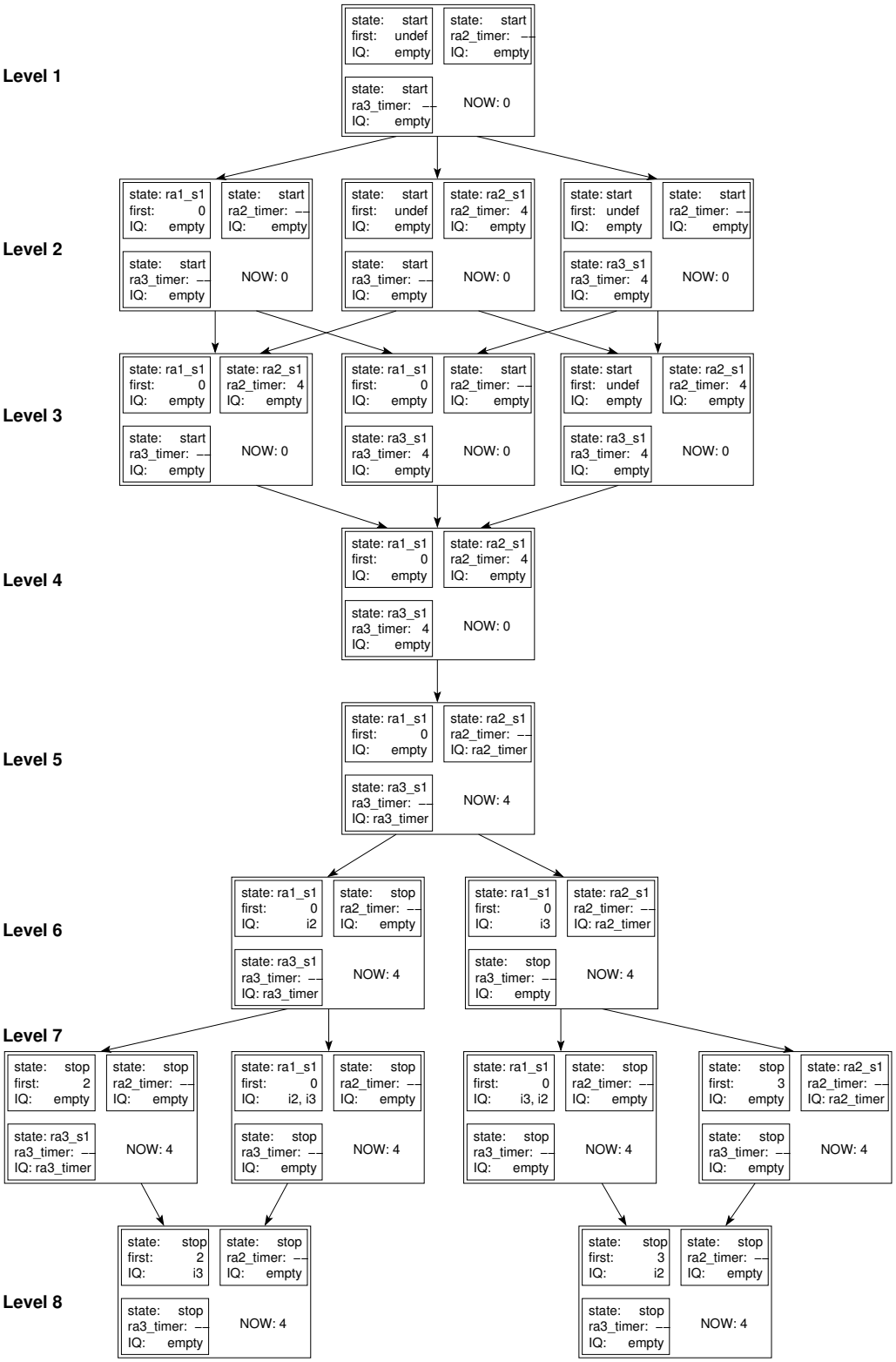


Figure 3.3.: Reachability Analysis Example - Reachability Graph.

The global state on level 4 is generated independently of the order of executing the start transitions.

The following transition (from level 4 to level 5) is implicit and represents increase of system time to the next point in time where an action is possible. For different time progress strategies, see section 5.1.3. At time four, the timers of *RA2* and *RA3* expire and are placed as (timer) signals in the respective input queues (level 5).

Thus, either the enabled transition of *RA2* or the one of *RA3* may fire, placing signal `i2`(left global state on level 6) respectively `i3` (right global state on level 6) in *RA1*'s input queue.

From level 6 to level 7, either *RA1* may consume the signal in the input queue, set variable `first` and terminate (first and third global state - from left to right - on level 7) or the other expired timer is consumed placing the second signal in the input queue of *RA1*. Finally, from level 7 to level 8, the last not yet terminated process fires its transition (the one that has not been fired from level 6 to level 7) resulting in all processes being terminated (state `stop`) on level 8. The value of variable `first` is now either two or three depending on the consumption/firing sequence of *RA2* and *RA3*.

Validation. During reachability analysis, the global states can be checked for pre-defined properties. This is termed "Validation". The properties usually describe undesired behavior of the modeled protocol. Thus, validation can be used to detect design faults in the protocol under investigation.

For the example of figure 3.3, it might be checked whether `first` is never set to three. This property is violated in two reachable global states: the rightmost one on level 7 and 8 each.

Classification of Exploration Techniques. Different exploration techniques used in reachability analysis can be distinguished. Typical classification criteria are

- exhaustive or partial analysis
- blind search or guided search

First, it has to be determined whether a complete (exhaustive) exploration of the state space can be performed, or a partial analysis strategy has to be applied. Many of today's protocols result in large models and thus in possibly huge state spaces where only a partial analysis can be performed within given limitations of available computation time and memory.

The most common strategies for exhaustive exploration are depth-first and breadth-first traversal. The highly memory-consuming nature of breadth-first traversal rules it out for large models. A depth-first strategy is introduced in section 3.2.1.

Both, breadth-first and depth-first traversal may also be applied as partial exploration strategies. If the reachability graph cannot be explored completely within given time and memory-limits, the exploration terminates prematurely.

Generally, partial exploration strategies may also be subdivided into two strategies: blind search and guided search. In contrast to blind search, guided search takes information about the model,

the application area or any other user-/modeler-provided information into account. The available information forms the basis for heuristic traversal algorithms. The algorithms contributed in part II are based on such heuristics.

The most common strategies for blind search are "Random Walk" and "Bitstate" (also known as "Supertrace") traversal. These two strategies are described in sections 3.2.2 and 3.2.3.

For all of the partial exploration algorithms it is difficult to estimate the portion of the state space that will be explored.

Challenges in Reachability Analysis. Reachability analysis and validation of large models poses several challenges. The main challenge is the inherent state space explosion problem. This problem can be tackled by two approaches that can also be combined:

- Application of state space reduction techniques, especially partial ordering techniques (see chapter 4) to reduce the state space without loss of "interesting" behavior.
- Using specialized heuristic partial exploration techniques (see chapters 6 and 7) to increase the chances of investigating mainly the "interesting" parts of the state space.

3.1.1. Application Areas

Validation based on reachability analysis is used for many purposes. Tools and algorithms have been developed or adapted to increase the performance for the specific validation goals. Nevertheless, tools for "general" (multi-purpose) reachability analysis are also available. The specialized tools often require a specific input language, for example SPIN [Hol97], HyTech [HHWT95], LUSCETA [Jon99]. While these tools are highly suitable for their respective purpose, it is usually difficult to validate any other properties as well. For example, tools specializing on software design testing will hardly provide means for performance analysis. One of the advantages of the (commercial) general tools is the use of standardized input languages [Tel01]. Furthermore, different analysis goals can be checked at the same time. The algorithms used in these tools are required to fit for any possible validation goals.

The list of analysis goals includes, but is not limited to:

- Performance Analysis (QUEST, SPEET)
- Testing Software Designs (χSuds,[Hol87a] , SPIN, VeriSoft, Flavers)
- Hardware Test-Cases (Aniseed)
- Verification of Protocol Properties:
  - Timing Properties (UPPAAL, HyTech, Kronos)
  - Safety Properties ([EN99])
  - Fault-Tolerance Properties (RAFT as a contribution of this thesis)
- General Reachability Analysis (SDT)

Tools focusing on the respective goal are given in parenthesis.

Table 3.1 summarizes some of the existing tools and algorithms for reachability analysis. The tools/algorithms contained in the table are only a representative subset of the many tools available. They are either wide-spread, based on a standardized language or focused on dependability goals.

Property → Tool ↓	**Language**	**Purpose**	**Literature**
QUEST	extended SDL	performance	[DHMC96]
SPEET	extended SDL	performance	[SL97]
χSuds	SDL	software	[LH00]
UPPAAL	own language	timing	[BDL04, LPY97b, LPY97a, BLL+96, BGK+96]
SPIN	Promela	software	[Hol97, JLS96, DM04, ELL01]
VeriSoft	C, C++	software	[God03, God97]
HyTech	own language	timing	[AHH96, HHWT97, HHWT95]
[Hol87a]	Argos	software	[Hol87a]
Kronos	own language	timing	[BDM+98, DY95, Yov97]
Flavers	Ada, Java	software	[CCO02]
Aniseed	SDL	hardware	[CT97]
SDT	SDL	general	[Tel01]
RAFT	SDL	fault-tolerance	[Böh05]
[EN99]	Petri Nets	safety	[EN99]

Table 3.1.: Exploration Tools.

Validation of Fault-Tolerance Properties. None of the existing algorithms or tools provides special heuristics for analysis of fault-tolerance properties except for the RAFT tool contributed in this thesis. Most of the specialized tools cannot be extended or adapted to include such algorithms either because their analysis goal differs too much from the fault-tolerance-validation goal, or they are proprietary. Some of the tools are not even maintained any more. Therefore, the performance of the novel algorithms cannot be compared to existing specialized algorithms, but to the standard general algorithms.

3.2. General Reachability Analysis Algorithms

In the last section different algorithms and tools have been discussed. It has been argued that none of them is really suitable for performing validation of large fault-tolerant communication protocols. Mainly due to the different validation goals of those tools it is impossible to compare the employed algorithms with the ones contributed in part II of this thesis. Therefore, the performance of the novel algorithms has to be compared to the general algorithms: exhaustive exploration, random walk and bitstate traversal. The comparison is provided in part IV. As

already discussed in the previous section, the SDT validator provides these algorithms. Furthermore, the input language of SDT is SDL, thus allowing for easy comparison between the novel algorithms and available implemented general ones. Therefore, the following description of the general algorithms is based on their implementation within the SDT tool.

3.2.1. Exhaustive Exploration

The exhaustive validation method performs a complete depth-first traversal of the reachability graph if time and memory restrictions allow for it. Once a reconvergence is detected on a path, that path is not pursued any further. A reconvergence in the reachability graph occurs if a previously generated global state is visited again. The exhaustive algorithm requires a large amount of main memory as all global states need to be kept. Global states in SDT are defined by the current local states of all processes, their variable values, active timers and input queues. Figure 3.4 shows an example snapshot during an exhaustive exploration. Black circles represent visited states, white ones states that have not been visited yet.

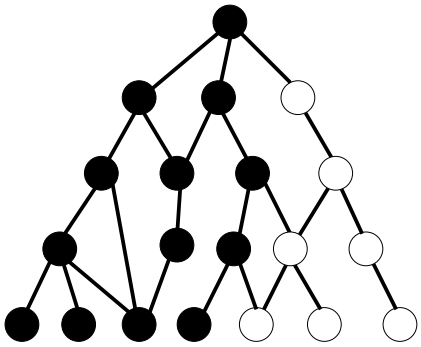


Figure 3.4.: Exhaustive Traversal.

3.2.2. Random Walk

The random algorithm as implemented in SDT resembles a set of simulation runs. A single path is generated (fig. 3.5). The path is created by randomly selecting the next enabled transition to be fired. The algorithm can be repeated for a specified number of times. Yet, there is no guarantee that each path is generated only once. Application of the random algorithm does not allow for conclusions about the portion of the state space that has been explored.

3.2.3. Bitstate Exploration

The bitstate algorithm is also known as supertrace algorithm. It has been introduced in [Hol87b, Hol88]. The bitstate algorithm as implemented in SDT is defined based on the exhaustive algorithm. Instead of comparing complete global states for reconvergence detection, a hash code is computed for each global state and only the hash codes are compared. Optimistically,

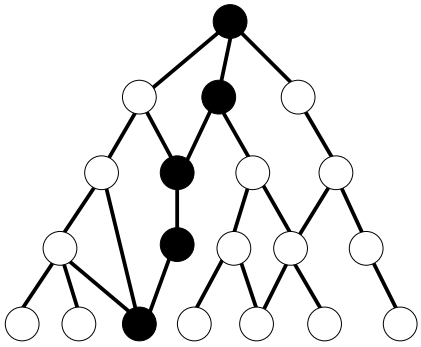


Figure 3.5.: Random Traversal.

identical hash values are assumed to be reconvergences. Figure 3.6 depicts an example snapshot during bitstate traversal. The numbers associated with the global states represent the respective hash codes for the visited states. Grey states marked with an "X" indicate that the path is not continued, because of an assumed reconvergence.

To decrease the chances for collisions in the hash table, SDT uses two hash tables with different hash functions. A reconvergence is assumed if both functions lead to a collision in their respective table.

The advantage of the bitstate algorithm with respect to the exhaustive algorithm is the reduced memory requirements. However, chances for an unjustified reconvergence detection, possibly resulting in missing important behavior, are inherent in the bistate algorithm.

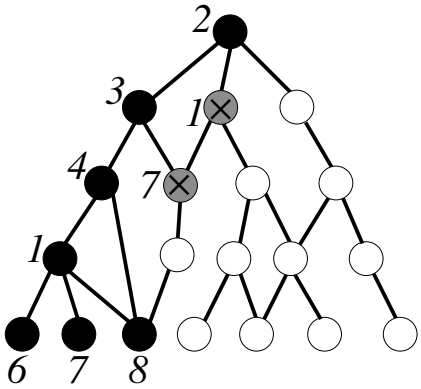


Figure 3.6.: Bitstate Traversal.

Precise information about the explored portion of the state space is not available for the bitstate algorithm. After completion of the reachability analysis, the only information is the number of detected reconvergences. If the number is low, it can be assumed that a major portion of the state-space has been explored. With an increasing number of collisions, the portion of the visited state space is presumably decreasing.

Part II.

Algorithms

4. Motivation and Introduction

State transition models of protocols have a long tradition, and substantial experience has been gained by using respective tools. Nevertheless, the state space explosion problem still exists: When n components are modeled by an automaton with s_i local states each, $i \in \{1, \ldots, n\}$, this yields a total of $s_L := s_1 + s_2 + \cdots + s_n$ local states and up to $s_{Gmax} := s_1 \cdot s_2 \cdot \cdots \cdot s_n$ global states generated by reachability analysis, where $s_{Gmax} \gg s_L$ can be an extremely large number preventing exhaustive exploration.

Typically, the global state space consists of less than s_{Gmax} states according to the event sequences defined by the model. Receive operations are executed always after the respective send operations, for example. Moreover, as far as the model comprises a notion of time, all events are ordered according to their points in time. This reduces concurrency and thus the state space. Models of fault-tolerant systems should generally express the elapse of time, because assumptions on timing, timed schedules and timeouts are fundamental elements of fault tolerance algorithms. In principle there are four types of time models:

- Constant timing: Durations of timed actions are constants to be specified before the model is executed for analysis. Constant timing is sufficient to express most timeouts. However, non-deterministic fault occurrences and fault effects cannot be modeled adequately. In contrast to this completely static solution the three following ones are dynamic.

- Variable timing: Durations of timed actions can be determined by the current contents of a data variable in the moment when the timed action begins. This approach is supported by SDL tools [DHMC96, Tel01] (although the SDL definition does not strictly enforce variable timing). Variable timing is appropriate for fault occurrence, but not all types of non-deterministic fault propagation. In section 5.1.3 the drawback will be discussed in detail.

- Interval timing: Durations Δ of timed actions are limited by two variables, a lower bound α and an upper bound β, such that $0 \leq \alpha \leq \Delta \leq \beta$ [EN99]. Note the difference to variable timing: Here, the duration Δ needs not be determined at the beginning of the timed action. Instead, at any time out of the interval $[\alpha, \beta]$ the timed action may terminate. Moreover, it can be forced to terminate prematurely by occurrence of some event. This way all behaviors of faults and their countermeasures can be expressed properly.

- Stochastic timing: Durations of timed actions follow a probability distribution. This is an important feature for quantitative analysis as known from timed Petri nets, for example [BK02], but not relevant for the verification of a fault tolerance algorithm. It must tolerate all specified (timing) faults regardless of their probability.

The modeled system behavior and the time model lead to a global state space consisting of s_{Gmod} states, which is typically much smaller than s_{Gmax}, yet still too large in many cases.

The global state space consisting of s_{Gmod} states cannot be changed without modifying the behavior of the system. Removing some states can lead to a subset, a superset, or simply a different set of potential behaviors − each represented by a path through the global state space. However, for some reduction techniques it can be shown that the change does not have an effect on the questions one is interested in when analyzing the model. The following methods of partial state space exploration are well-known:

- Partial ordering [ABH+97, CGP99]: In addition to the ordering of events by the model itself a partial order among the events is defined such that the analysis only follows a single "representative" sequence of events out of a set of sequences. Assume two sequences of local state-entering events: (a_1, a_2, a_3) in node A, and (b_1, b_2) in node B. Normal reachability analysis provides all "mixtures" of the concurrent nodes A and B:

  $$\begin{array}{llll} (a_1,a_2,a_3,b_1,b_2), & (a_1,a_2,b_1,a_3,b_2), & (a_1,b_1,a_2,a_3,b_2), & (b_1,a_1,a_2,a_3,b_2), \\ (a_1,a_2,b_1,b_2,a_3), & (a_1,b_1,a_2,b_2,a_3), & (b_1,a_1,a_2,b_2,a_3), & (a_1,b_1,b_2,a_2,a_3), \\ & (b_1,a_1,b_2,a_2,a_3), & (b_1,b_2,a_1,a_2,a_3). & \end{array}$$

  However, in the absence of interactions between A and B, one is not interested in all these sequences. A single representative, $(a_1, a_2, a_3, b_1, b_2)$ for example, is sufficient. The model checking tool can generate this sequence by simply processing node A first, and node B then, until an interaction takes place. Note that in this context time consumption has to be considered an interaction as well. If, say, a_3 follows a_2 after a duration of 2 model time units, and the remaining events are not timed, then only the following sequences are possible:

  $$\begin{array}{lll} (a_1,a_2,b_1,b_2,a_3), & (a_1,b_1,a_2,b_2,a_3), & (b_1,a_1,a_2,b_2,a_3), \\ (a_1,b_1,b_2,a_2,a_3), & (b_1,a_1,b_2,a_2,a_3), & (b_1,b_2,a_1,a_2,a_3). \end{array}$$

  After applying partial ordering $(a_1, a_2, b_1, b_2, a_3)$ can be taken as representative. The model checker obtains it by passing from node A to node B in the moment when A consumes time after a_2.

  A partial ordering algorithm based on single fault regions is introduced in section 5.1.

- Heuristic selection: If one has knowledge which paths are likely to answer the questions one is interested in, one can skip the remaining paths. This technique can be applied whether the knowledge can give guarantees or only provides "guesses". Algorithms based on heuristics are introduced in chapters 6 and 7.

- Stochastic selection: One can also select paths at random (which may come close to a simulation) or by pseudo-random techniques like the bitstate algorithm (see section 3.2.3, for example): Global states are encoded by a hash function, and collisions in the hash table are not resolved. Then the size of the hash table (a simple bit vector) defines an upper bound to the explored state space. Depending on the hash function a number of paths is skipped. Stochastic selection can be combined with all other reduction methods.

- Path cut: Most questions to be answered by the model have a decision point in time or a decision point in the event sequence. The behavior thereafter is not relevant for answering the question. If the decision point is known one can cut a path there. This technique requires knowledge about the concrete model.

By applying any method of partial state space exploration, or a combination thereof, a smaller global state space is obtained with only $s_{Gpartial}$ states, typically $s_{Gpartial} \ll s_{Gmod} \ll s_{Gmax}$.

Chapter 5 focused on several state space reduction techniques including the SFR-PO (single fault region partial ordering) approach, a major contribution of this thesis. Chapters 6 and 7 introduce two novel algorithms for heuristic partial state space exploration as another major contribution. These algorithms also exploit, to a certain extent, the knowledge that the modeled systems represent fault-tolerant protocols.

5. State Space Reduction Techniques

Throughout this chapter, three state space reduction techniques are presented. These techniques are specifically tailored to fault-tolerance protocols. In section 5.1, a novel approach to partial ordering based on single fault regions is presented. Section 5.2 is dedicated to special handling of the start transitions in SDL. By specification of special processes, the state space can be reduced further. These processes are introduced in section 5.3.

5.1. Single Fault Region Partial Ordering

In this section, a novel, special variant of partial ordering exploiting the nature of models of fault-tolerant systems based on single fault regions (SFR-PO) is introduced. It has also been published in [BE04].

In section 5.1.1 the basic ideas of a novel approach to fault-region-related state space reduction are presented. Sections 5.1.2 and 5.1.3 discuss the realization of the principles in SDL and present a solution how they can be implemented without any modification of SDL.

5.1.1. State Space Reduction Based on Single Fault Regions

Usually fault-tolerant systems are subdivided into fault regions [EN99, Ech84, Kes02]. The meaning of a fault region here is a set of components the fault-tolerance algorithm considers either fault-free or faulty as a whole. For fault-tolerant operation it is not necessary to locate a fault within a fault region.

In the literature similar, but not identical terms like "fault containment region" and "smallest replaceable unit" have been defined [Lap92]. In contrast to the widely-used understanding of "fault containment regions" or "smallest replaceable unit" etc. fault regions need not be disjoint. Figure 5.1 shows an example with 8 components A, B, C, D, E, F, G and H. Components are represented by circles. The rounded rectangles enclose the components of one fault region each. Boxes indicate the single fault regions. For a fault tolerance algorithm it can be sufficient to locate a fault in either of the three overlapping fault regions $fr_1 = \{A, B, C\}$, $fr_2 = \{C, D, E, F\}$ or $fr_3 = \{E, F, G, H\}$.

To obtain disjoint sets of components, all intersections and differences of fault regions are defined *single fault regions*. In figure 5.1 there are five single fault regions:

$$\begin{aligned} sfr_1 &= \{A, B\} = fr_1 \setminus fr_2, \\ sfr_2 &= \{C\} = fr_1 \cap fr_2, \\ sfr_3 &= \{D\} = fr_2 \setminus fr_1 \setminus fr_3, \\ sfr_4 &= \{H, G\} = fr_3 \setminus fr_2, \\ sfr_5 &= \{F, E\} = fr_2 \cap fr_3. \end{aligned}$$

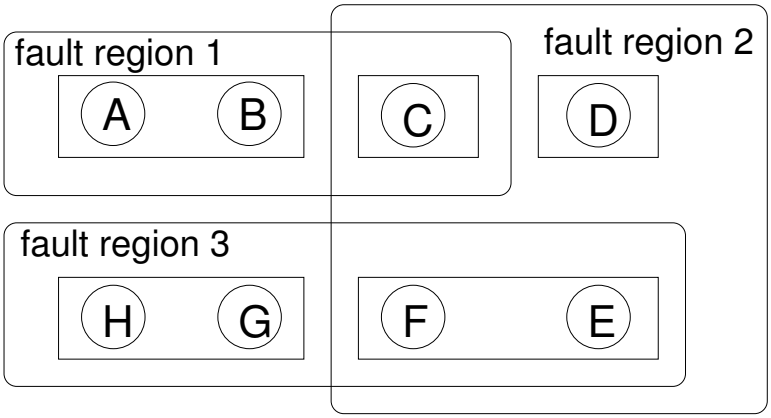


Figure 5.1.: Example for (Single) Fault Regions.

From the fault tolerance viewpoint a faulty single fault region is considered a single fault, whereas a faulty fault region may represent a multiple fault (stretching over several single fault regions). If and only if all faulty components are elements of one fault region the system must tolerate the faults.

Reasonably the interfaces between single fault regions are equipped with countermeasures against errors. The receiving single fault region checks incoming information in various ways before processing it. Corrupted information must not corrupt the receiving single fault region itself. Otherwise fault propagation could affect more components than covered by one fault region, before appropriate countermeasures can be taken. Thus fault tolerance might be lost.

The sceptical checks of incoming information are important or represent even the central parts of a fault tolerance algorithm. They include tests for consistency, correct encoding and plausibility as well as comparison between two, or voting among three incoming pieces of information. Naturally these operations take some time to be performed. When adding the time of the information transfer between the single fault regions, the time consumption is even higher. Even a minimum delay comparable to those of internal operations within a single fault region gives justification for the following approach:

SFR-PO "Single Fault Region Partial Ordering": Define single fault regions in the model of a fault-tolerant system. During reachability analysis apply partial ordering of the single fault regions, such that only one representative order is explored − as a substitute of all the concurrent event sequences among the single fault regions.

The approach can also be expressed in an operational way by introducing the following rule on the sequence of transition firing: For each point in model time state space exploration processes the single fault regions locally one after the other. Concurrency within a single fault region is fully considered. However, different execution sequences among the single fault regions are excluded from state space exploration. If there are no more local transitions to fire at the given point in model time, the interaction transitions between the single fault regions with zero time consumption are executed (if there are such transitions at all). Then, state space exploration passes to the next point in model time and processes the respective time-consuming transitions. These can be interactions between single fault regions and/or local timers. The new point in

time can be determined by any of the time models (constant timing, variable timing, or interval timing) except stochastic timing. Then, the described cycle is repeated for the current (new) model time, and so on.

SFR-PO leads to three types of transitions:

- LIT: Local immediate transitions within a single fault region. These transitions are not time-consuming. Typically they model fast local decisions and operations.
- LTT: Local time-consuming transitions within a single fault region, to express longer lasting operations or timeouts. In most cases the durations are constants (which can also be expressed by variable timing and interval timing, of course). Constant durations are a big benefit for the state space, because the resulting global event sequences are unique rather than affected by concurrency. This keeps the global state space smaller.
- GT: Global interaction transitions between single fault regions, to express transfer of information. The induced synchronization is only single-sided. The receiving single fault region has to wait for the sending one, not vice versa. Double-sided rendezvous synchronization can be achieved by mutual GT's.

Because of their time consumption LTT's are executed after all the LIT's which are able to fire at the current point in model time. This ensures the correct order of firing. For the correct firing of the GT's two alternatives exist:

- Lower priority of GT's: Then the higher priorities of the LIT's make them firing first, as desired. This approach allows for both time-consuming and non-time consuming GT's. The latter may be interesting, if the whole time model is relative coarse (only long-lasting operations and timers consume model time, for example). The lower priority of the GT's expresses a non-quantified small processing time. If the model or the tool for analysis does not support priorities, this alternative cannot be taken.
- Non-zero time consumption of GT's: All GT's consume some model time. If communication delays are included, the duration is typically non-deterministic (expressed by variable timing or interval timing). Time consumption makes the GT's fire after the LIT's, as desired. By selection of an appropriate time resolution any relationship between durations of LTT's and GT's can be achieved.

Model analysis following the SFR-PO approach covers the behavior of the real system completely, if the following SFR-PO condition holds: Any local sequence of any number of LIT's executes faster than any GT. At first sight this seems to be far from reality. A simple counterexample can be constructed by taking a local sequence of 1000 LIT's. It is likely to take longer than a single GT. Only for short sequences of very few LIT's the SFR-PO condition may be satisfied.

The SFR-PO condition has to be taken the other way round: It is the responsibility of the modeler to make sure the local sequences are sufficiently short. If not, a time consuming LTT must simply be inserted in the sequence. This is not a burden to modeling. Moreover, it is a general improvement to the accuracy of the model. Having a long LIT sequence without model

time consumption means a deviation between the model and reality. Such deviations should be avoided, even when fault tolerance is not an issue. Consequently, good models tend to satisfy the SFR-PO condition anyway.

For the remaining discrepancies violating the SFR-PO condition it is a practical question whether or not the modeler considers it a big burden to modify the model accordingly. However, many fault-tolerant systems are already designed very close to the SFR-PO condition. Generally, the SFR-PO condition does not restrict modeling power, because LTT's can always be inserted.

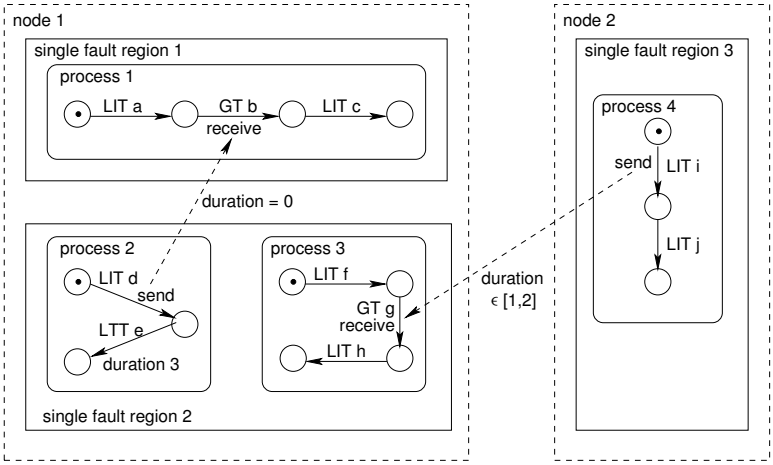


Figure 5.2.: Example for SFR-PO.

The example depicted in figure 5.2 shows a system with two nodes (dashed rectangles). The first node consisting of two single fault regions (boxes), the second node consisting of one single fault region. Single fault region 2 in node 1 comprises two processes (rounded rectangles). The other single fault regions contain one process only. States are represented by circles - initial states of the processes are marked by a dot. Arcs between the states indicate transitions and signal flow is represented by the dashed arcs.

Single fault region 1 is processed first (starting with transition a) up to the point before GT b. Then state space exploration continues with single fault region 2. Here, we have to consider concurrency among processes 2 and 3. So far, the beginnings of two paths (a, d, f) and (a, f, d) have been generated. Now the exploration of processes 2 and 3 is interrupted because of LTT e and GT g, respectively. When exploration continues with single fault region 3 the two paths are extended to (a, d, f, i, j) and (a, f, d, i, j). Then the exploration comes back to single fault region 1 and processes the low priority receive operation b as well as the immediate transition c. Now, all transitions are unable to fire at the current point in model time. Hence, time advances to $t \in [1, 2]$, to be expressed symbolically, see the last paragraph of section 5.1.3. Then g and h fire, leading to $(a, d, f, i, j, b, c, g, h)$ and $(a, f, d, i, j, b, c, g, h)$. Finally LTT e fires after the remaining duration $3 - t$ has elapsed.

By pure node-oriented partial ordering ten paths would have been obtained instead of just two. The ten paths combine the six permutations of a, d and f with the concurrency of f and (b, c). Without any kind of partial ordering, the number of paths would be 210 because (i, j) are concurrent to (a, b, c, d, f). Even in this simple example, SFR-PO yields a paths reduction factor

of $\frac{210}{2} = 105$ (when compared to standard state space exploration) or $\frac{10}{2} = 5$ (when compared to node-oriented partial ordering).

In the presence of faults, SFR-PO exhibits advantages as well. When assuming a timing fault in single fault region 3, the receiving transition g can fire prematurely, in time, or too late. The processing of the wrong timing is encapsulated in single fault region 2 as far as possible with respect to timing. In all, eight paths are generated (transitions i and j in the faulty unit are not of interest here):

$$\begin{aligned}
\text{premature}: \quad & (a,d,f,g,h,b,c,e), \quad (a,f,d,g,h,b,c,e),\\
& (a,f,g,d,h,b,c,e), \quad (a,f,g,h,d,b,c,e),\\
\text{in time}: \quad & (a,d,f,b,c,g,h,e), \quad (a,f,d,b,c,g,h,e),\\
\text{too late}: \quad & (a,d,f,b,c,e,g,h), \quad (a,f,d,b,c,e,g,h).
\end{aligned}$$

Node-oriented partial ordering would have generated 32 paths in this case.

In the following section, concepts are developed to apply the SFR-PO method to SDL models. For transitions of type LTT and GT efficient solutions are presented which fully conform to SFR-PO on one side, and do not add too much complexity to the model and run-time to the analysis on the other.

5.1.2. Solutions for SDL

The SFR-PO approach requires single fault regions to be specified in the model. This section describes how single fault regions and time consumption between them can be implemented in SDL without extending the language.

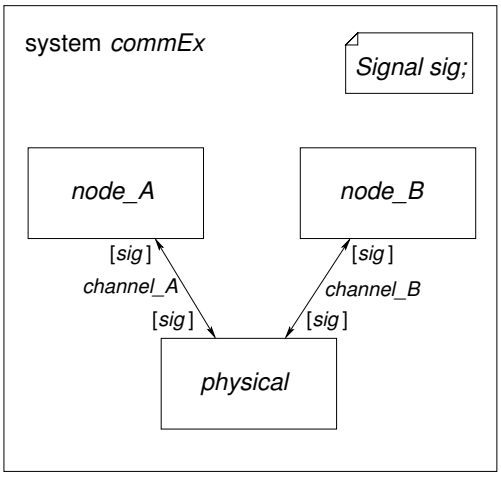
Single Fault Regions in SDL.

The hierarchical structure of an SDL model as described in section 2.2 is quite similar to the one shown for the single fault region example in figure 5.2 (page 40). At the lowest level, processes with states and transitions are defined. They represent a functional unit like an electrical bus driver.

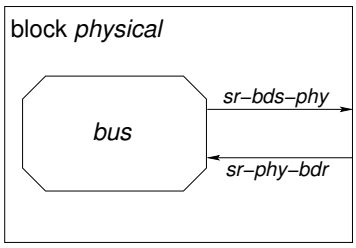
The concept of `blocks` in SDL enables the user to easily structure the model according to the required single fault regions. Processes belonging to the same single fault region can be grouped into a `block`. In the example depicted in figure 5.3 each node (`node_A`, `node_B`) is assumed to be a single fault region which may become faulty as a whole. The `physical` block is modeled to represent another single fault region containing only the bus process. For the example presented in section 5.1.1 (fig. 5.2, page 40) the three single fault regions would be implemented as a block each.

SDL provides 3 types of transitions:

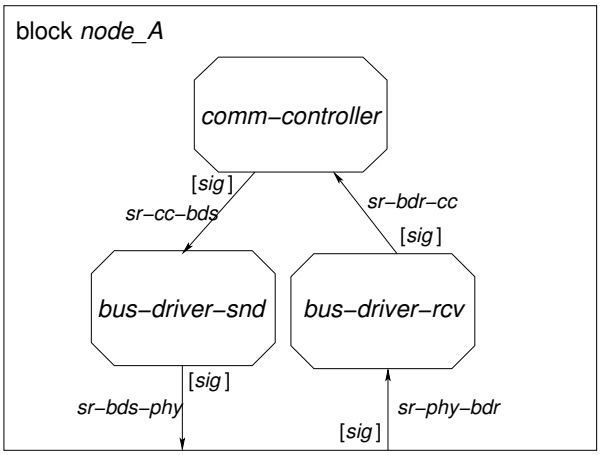
- transitions enabled through signal reception (corresponding to the LIT's)



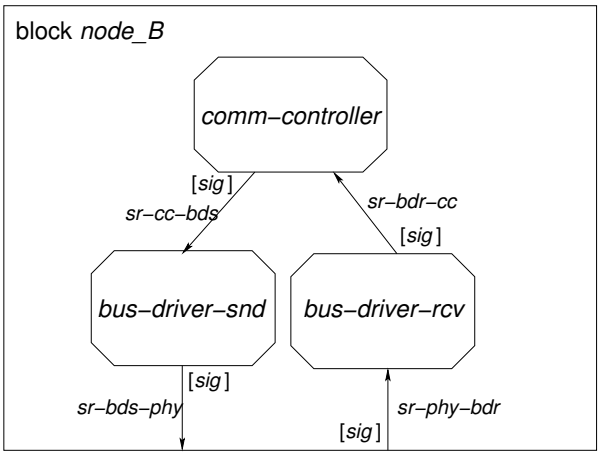
(a) System With Blocks



(b) Block Physical With Bus Process



(c) Block Node_A



(d) Block Node_B

Figure 5.3.: Example SDL System.

- transitions enabled through expiration of a timer (corresponding to the LTT's)
- spontaneous transitions (corresponding to the LIT's as well)

GT's are not explicitly available in SDL, but can be seen as transitions enabled through signal reception across the borderline between two blocks.

Conclusion: The SFR-PO approach can be realized in SDL by exploiting the `block` concept.

Modeling Time Consumption Between Single Fault Regions in SDL.

In section 5.1.1 the necessity for modeling time consumption between single fault regions has been motivated. SDL does not provide special means for modeling signal delays between processes, whether located in the same block or crossing block boundaries.

As already described in section 2.2, signals between processes are sent via `signal routes` indicated by the prefix *sr* in the example shown in figure 5.3. `Signal routes` do not support any delaying at all, thus communication via `signal routes` requires zero-time.

For communication of processes located in different blocks, `signal routes` need to be connected to `channels` which allow communication between blocks (see also sec. 2.2). Each channel can be defined to impose either zero-time delay on the signals, or to delay each signal for "an indeterminant and non-constant time interval" [ITU93b]. Thus no *specific* limits on the delay can be defined.

Throughout the remainder of section 5.1.2, a method of implementing well-defined delays in SDL models is presented. The general idea is to implement *delay processes*. These processes are normal SDL processes with the only functionality of time consumption. Once the basic model has been defined, it is possible to create all necessary delay processes automatically.

Communication delays between two entities (nodes, for example) can always be expressed through three different delays:

- **DS:** Delay at the sender side (equal for all outgoing signals)
- **DR:** Delay at the receiver side (equal for all incoming signals)
- **DT:** Delay during transmission (depending on each pair of sending and receiving nodes)

The overall delay **DO** induced on a signal being sent from node i to node j can be calculated as $DO_{i,j} = DS_i + DT_{i,j} + DR_j$, where each delay may be zero.

Several kinds of implementing the delay processes are possible:

- **DO-processes:** In some cases it is sufficient to place one delay process expressing the sum $DO_{i,j}$ between each sender-receiver pair. Note that $DO_{i,j}$ is not necessarily equal to $DO_{j,i}$. DO-processes can be applied if DS_i and DR_j are constant.

- **DS-processes:** Combining the three delays in one process is not always possible. In many cases the delays on the sender side are subject to a small jitter. Thus the signal issued by a sender i at time t_1 is forwarded to the communication layer at time $t_2 \in [t_1+x, t_1+y]$. Thus, $DS_i \in [x, y]$. The jitter is equal for all recipients. This dependency can only be expressed if the decision on the value for t_2 is taken in a single process. Including this decision in every DO-process between node i and its respective recipients may yield different values chosen for t_2.

- **DR-processes:** If the receiver delay may change over time, these changes must be considered for all incoming signals. Implementing this in each DO-process would be difficult. Thus an extra DR-process is preferable.

- **DT-processes:** If $DT_{i,j} > 0$, then a process has to be provided implementing this delay. For $DS_i = 0$ and $DR_j = 0$ the DT-process is a special case of a DO-process.

- **Combinations:** All combinations of DS-, DR-, and DT-processes may be implemented.

If automatic generation and inclusion in a model is pursued, then the preferred solution should be to provide a DS-process for every sender, a DR-process for every receiver and a DT-process between every sender-receiver pair (in each direction). This minimizes the specification effort.

So far, the general approach of how to position the delay processes within a system has been discussed. Each of these delay processes may be subject to constant timing, variable timing or interval timing as introduced in chapter 4.

Common to all three timing cases is the implementation through `timers`. SDL also provides arrays of timers. Arrays of timers are especially useful if delays of signals may be overlapping.

The following two paragraphs show how the three kinds of timing can be implemented in SDL.

Variable Delay Process Implementation.

Figure 5.4 shows an example implementation of a delay process with variable delay. In the model an array of timers *t_sig1* is defined. i is a counter indicating the first free position in the timer array. i is initialized to zero during the start transition. The process will then remain in state `wait` until *signal1* has been received (left side). *signal1* contains the parameter x indicating the duration of the delay. The first timer *t_sig1(0)* is set to x time units from the current time ($now+x$) and i is incremented. We use a modulo 100 counter here. The value for the modulo operation has to be an upper bound of concurrently active timers. To determine this upper bound is the responsibility of the modeler. After setting the timer, state `wait` is entered again. Now, more signals may arrive resulting in the same transition firing again (resulting in overlapping durations), or one of the timers may expire. In the latter case, the input *t_sig1(o)* leads to firing the right hand side transition. o is a variable indicating the array position of the expired timer. This variable is not needed in the example implementation presented here, but is indispensable if different signals are delayed by the same process (see next paragraph). Expiration of a timer leads to an output of the signal *signal1* with the delay parameter set to zero. Then, state `wait`

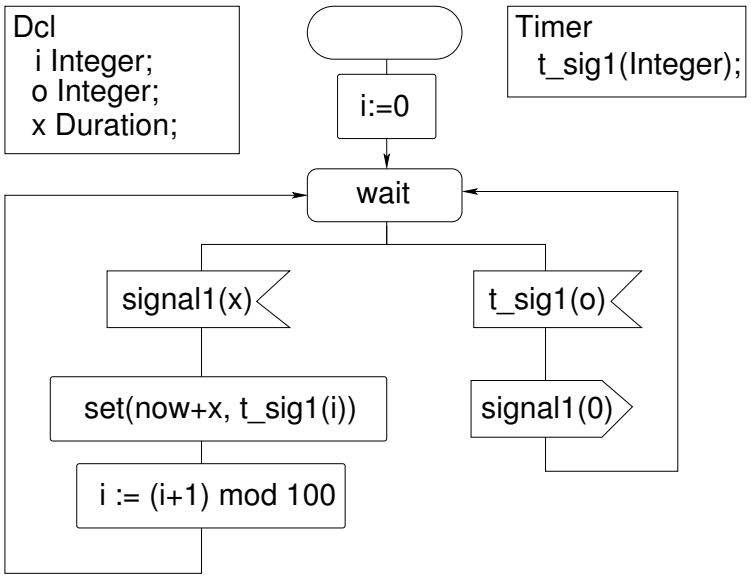


Figure 5.4.: Example of SDL Delay Process (Variable Delay).

is entered again. And so on.

If there are several different input signals to the delay process, an array containing the signal and its parameters has to be implemented as well. If the index of the incoming signal and its respective timer are equal, the correspondence between a timer and its signal is preserved. If a timer *t(o)* expires, *o* indicates the position in the signal array containing the corresponding signal. This signal will then be forwarded to the receiving process.

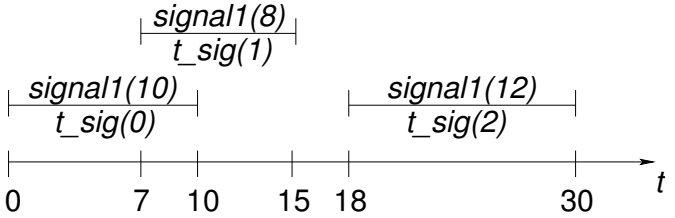


Figure 5.5.: Example for Variable Delay.

Figure 5.5 shows an example for three timers *t_sig(0)*, *t_sig(1)*, *t_sig(2)*. Timer *t_sig(0)* is set to 10 time units at time zero. Thus it expires at time 10. At time 7, another signal arrives requesting a delay of 8 time units. Thus, timers *t_sig(0)* and *t_sig(1)* are both active during time 7 to 10. *t_sig(1)* expires at time 15. After both timers expired, another timer *t_sig(2)* is set to expire 12 time units later.

Constant delays for all signals can be expressed by either setting x to the same value for all input signals, or to omit this parameter. In the latter case the value of x can be defined as a

constant. In figure 5.6, x is set to 10. The parameter associated with *signal1* is omitted. Again,

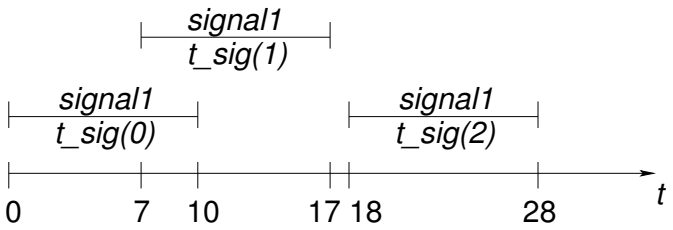


Figure 5.6.: Example for Constant Delay.

t_sig0 and *t_sig1* are overlapping for three time units.

Interval Delay Process Implementation.

SDL does not provide special timers for implementing interval timing. However, an implementation of such a delay process is possible by using spontaneous transitions (reserved word `NONE`). A spontaneous transition may fire at any time (or never) if it is activated (see also section 2.3.2). The definition of a spontaneous transition makes it a good candidate for solving the problem of specifying intervals.

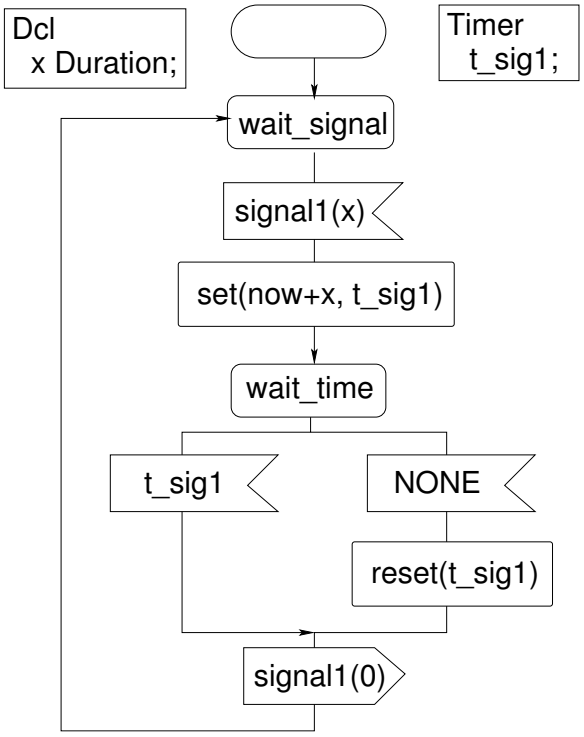


Figure 5.7.: Example of Interval Implementation in SDL.

A delay process representing interval timing can be modeled as depicted in figure 5.7. To focus on the relevant idea, signal overlapping is excluded from this example.

In state `wait_signal`, the process waits for the input *signal1* with the parameter specifying the upper bound x of the interval. The lower bound, in this example, is always assumed to be the current time. After reception of *signal1*, the timer is set as in the previous example and state `wait_time` is entered. In this state, either the spontaneous transition ($NONE$) fires at an arbitrary time, or the timer *t_sig1* expires. In both cases the signal is forwarded to the receiver and state `wait_signal` is entered again. If the spontaneous transition has fired, the timer is reset. This process implements the delay interval $[now, now + x]$.

The interval $[now + x, now + y]$, $0 < x < y$ can be implemented by adding (prepending) part of the model for a delay process for variable timing as presented in the previous paragraph.

Again these processes can be generated automatically in a preprocessing step.

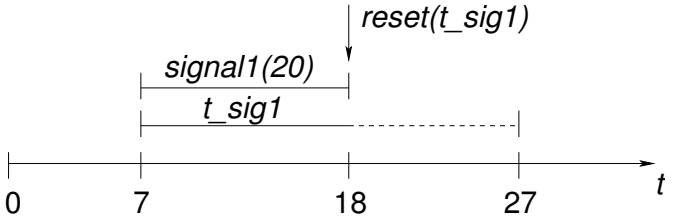


Figure 5.8.: Example for Interval Delay.

Figure 5.8 shows an example of an interval delay for signal *signal1*. *t_sig1* is set to the indicated 20 time units representing the upper bound of the delay. The spontaneous transition fires at time 18 resetting the timer. Thus, *signal1* has been delayed for 11 time units.

In conjunction with the mapping of single fault regions to the SDL hierarchy at the beginning of this section, delay processes substantiate the suitability of SDL for modeling state space reduction based on single fault regions.

5.1.3. Time Progress in State Space Analysis of SDL Models

Modeling delays, especially interval delays as presented in section 5.1.2, state space analysis should consider all possible points in time where the firing of a spontaneous transition generates a new path.

However, the SDL semantics of spontaneous transitions as described in section 2.3.2 (page 17) does not enforce this behavior. Instead, it allows all cases from not considering the spontaneous transitions at all to considering its activation at each simulation point of time.

Time Progress By Timer Expiration.

The approach of "time progress by timer expiration" is implemented in most tools, for example in SDT. It results in a specification conformant, but somehow deprecated state space. In this section, the approach will be discussed in detail and an example of the limitations will be given. Two ways to overcome these shortcomings will be presented.

A validation run starts at time $now = 0$ and executes all transitions that may fire (including spontaneous transitions) at time zero. If no transition is enabled anymore, the earliest expiration time of all currently active timers in the system (ET_{min}) is determined and $now := ET_{min}$ is set. As already discussed, timers in SDL are deterministic. Then all transitions (including the timers!) enabled at that time are executed, and so on.

For spontaneous transitions this non-continuous progress of time implies that they may only fire at those simulation times when - somewhere in the system - a timer expires. Thus it is possible that some execution paths are not generated when validating the system.

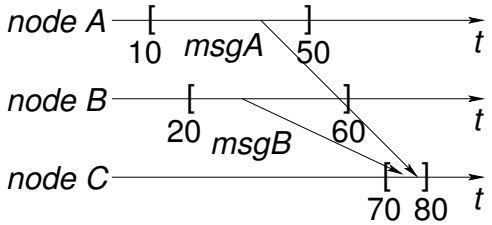
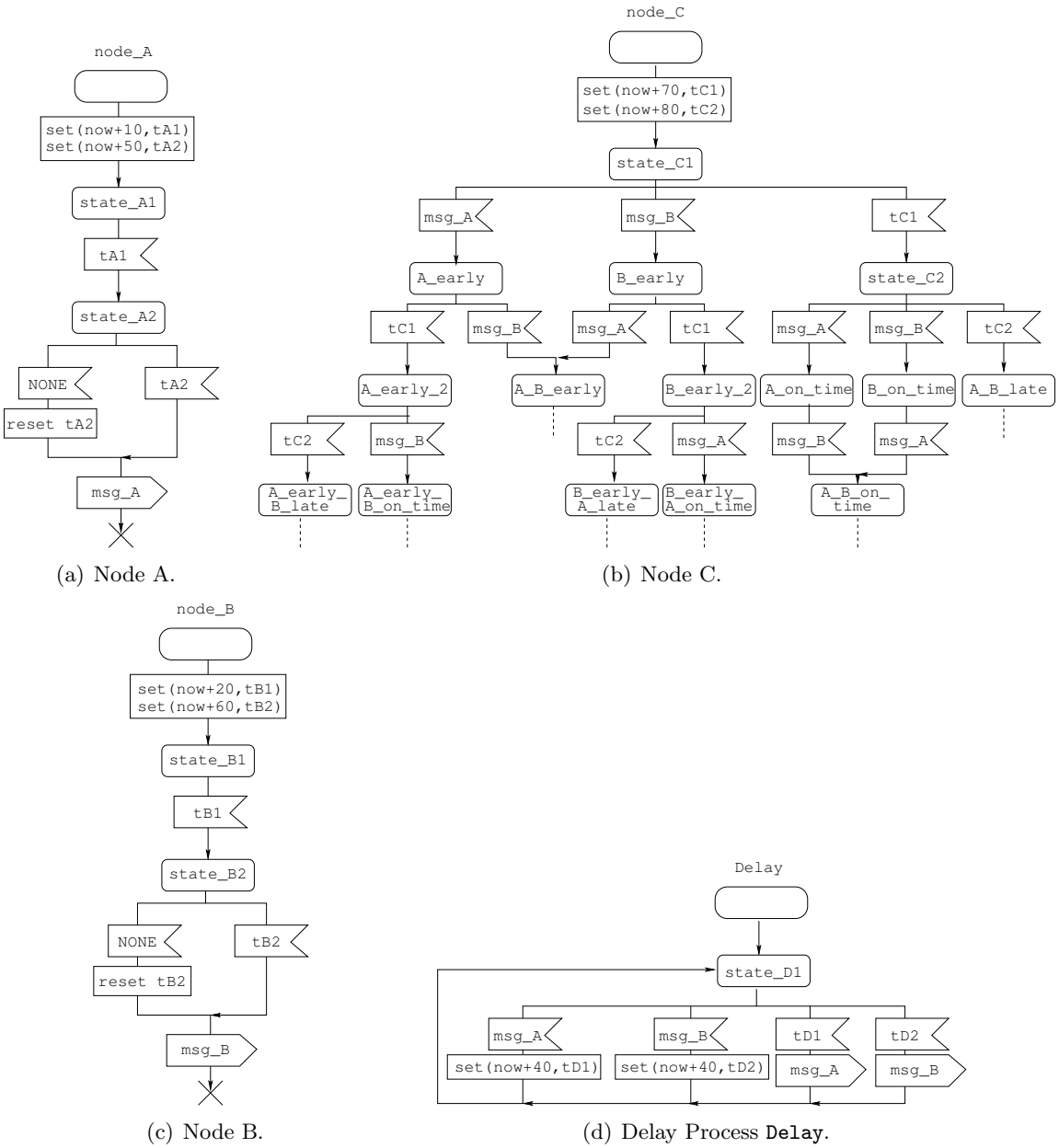


Figure 5.9.: Timing Problem Example.

Figure 5.9 shows a small example of interval timing. It depicts 3 nodes A, B and C over the simulation time *t*. `node A` and `node B` are supposed to send message `msgA` respectively `msgB` to `node C`. Message delay for both of the messages is assumed to be uniformly set to 40 time units. In the fault-free case, `msgA` and `msgB` are sent between time 30 and 40 and are thus expected to arrive at `node C` between time 70 and time 80. In case `node A` is faulty, it may send in the interval between time 10 and 50. `node B` may send in the interval between time 20 and 60 if it is faulty. Depending on the sending time of the messages all combinations of "msgA/B too early", "msgA/B on time", "msgA/B too late" are possible in the presence of faults.

The SDL diagrams for the 3 nodes and a delay process `Delay` are shown in figure 5.10. The model of the process implementing `node_C` is cut at the point where either both messages have been received, or the upper bound of `C`'s reception interval has been reached. The naming of the states appearing as leaves indicate the timing (`early`, `on time`, `late`) for each of the messages.

Following the discussion about time progress by timer expiration, only the points in model time 0, 10, 20, 50, 60, 70, 80 are considered for transition firing. Times stemming from delay process `Delay` depend on the time values chosen for `node_A` and `node_B`, respectively, and are included



(a) Node A. (b) Node C.

(c) Node B. (d) Delay Process `Delay`.

Figure 5.10.: Timing Problem Example - SDL Code.

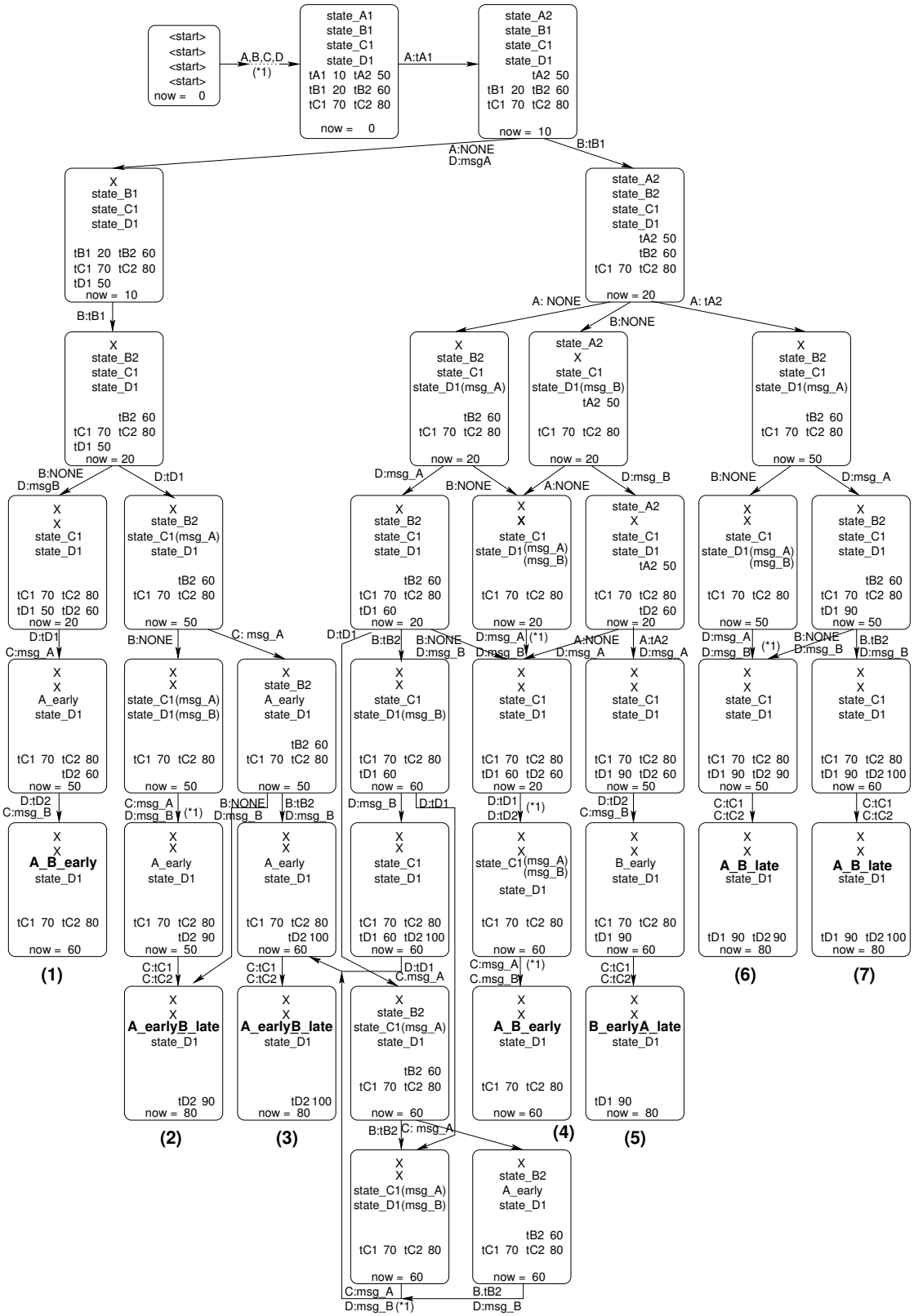


Figure 5.11.: State Space Generated with Time Progress by Timer Expiration.

in the following brief walk-through of the state space.

Figure 5.11 shows the state space generated by applying this time progress strategy. Each node shows the states of processes `node_A`, `node_B`, `node_C` and `Delay` in the first 4 rows. Then all active timers and their respective expiration times are depicted. *now* indicates the current simulation time. The states are connected by arcs labeled by the input(s) that are consumed in the transition. Arcs labeled with more than one input represent two consecutive transitions where the intermediate states are not depicted for readability reasons. If marked by ($*1$), they may be executed in arbitrary order. The interesting states are the leaves (marked (1)-(7) in figure 5.10(b)). They show the complete list of states process `node_C` might "end" in. States (1) and (4) represent the case where both messages have been received before the interval specified in node `node_C`. States (6) and (7) indicate that both messages have arrived after the interval. In the remaining states (2), (3) and (5) one of messages has been received before the interval and one after the interval.

The paths leading to `node_C`'s final local states `A_early_B_on_time`, `B_early_A_on_time` and `A_B_on_time` are not generated if time progress is implemented by jumping from timer expiration to timer expiration. In other words: No signal reception within the interval is discovered.

The two time progress strategies in the following paragraphs introduce solutions for time progress that will generate all possible paths.

Time Progress by a Ticker Process.

If progress of time can only occur if a timer expires, a straightforward solution to consider every point of time is to implement a timer that expires every time unit. This is preferably done in an extra process here termed "ticker process". As soon as the timer expires, it is set again. It's next expiration time is set to $now + 1$ and so on. By "1" the shortest non-zero duration according to the time granularity of the model is expressed.

The ticker process does not communicate with any of the other processes and is identical for every system. Thus it can be automatically inserted into the model in a preprocessing step.

This solution has the advantage that it can be used together with existing tools. For example the state space analysis performed by SDT will generate all possible paths if a ticker process is present.

The disadvantage of this approach lies in the large increase of the state space by many unnecessary paths. For most points of time it is irrelevant at which one the spontaneous transition fires. This state space explosion can be slightly reduced by several rules (R_i):

- **R1**: "tickering" only if spontaneous transitions are active;

- **R2**: "tickering" in steps > 1 time unit by setting it to (now+$stepSize$);
- **R3**: "tickering" is done in non-equidistant intervals;
- **R4**: sophisticated "tickering" by using already existing timers.

R1 requires that the ticker process is aware of enabled spontaneous transitions. For **R2** and even more for **R3** knowledge of the model is required. **R4** is highly dependent on the model − setting additional timers for ticks may be omitted if an already existing timer expires at the time the tick-timer would expire. Thus, tickering can be reduced by a large amount if already existing timers in the model are considered. If it is already known that, while a spontaneous transition is enabled, timers expire anyway, it may even be unnecessary to employ additional "tickering". Since **R2** to **R4** are model-dependent, it is not possible in general to insert such a ticker process automatically.

For small models, the ticker process is a suitable solution. For more complex models it may be too difficult to prevent the state space from growing too large. A combination of **R3** and **R4** has been successfully implemented for the large FlexRay [Fle02] model (see section 10.7). Timers have been adjusted to follow the communication schedule of a time-triggered protocol.

Time Progress by Symbolic Solution.

The symbolic processing of time does not require tickering. Instead, all potential time behaviors are covered by inequality formulae on time variables [EN99]. The basic idea can be illustrated by a small example. Assume that in figure 5.2 (page 40) the duration of the delay D_g between LIT i and GT g is not in $[1, 2]$, but in $[1, 4]$. Then LTT e with its delay $D_e = 3$ may fire either before or after GT g. LTT e firing before GT g can be expressed by attaching the inequality $D_e < D_g$ to the resulting state during reachability analysis. LTT e firing after GT g (or simultaneous to g) will have attached inequality $D_e \geq D_g$. During analysis the number of inequalities to be solved in a state will grow. If there exists no solution to the set of inequalities, the state cannot be reached and will not be included in the reachability graph. This way of processing time has been generally introduced in [EN99]. It ensures a complete coverage of all possible simulation points of time. The drawback is that worst case computation times for solving large systems of inequalities grow exponentially. Yet, the system of inequalities can be solved efficiently in nearly all cases.

The different timing models, and the ability to express any kinds of delays in SDL, in conjunction with expressing single fault regions through SDL-`blocks` shows the applicability of the SFR-PO mechanism in practice.

5.2. Start Transitions

The state space can be further reduced by imposing an order on the start transitions. Start transitions have been described in section 2.3.1 on page 15.

SDL process models always begin with an initial `start` state. Transitions from `start` states are handled slightly different than transitions between "normal" states. Start transitions, mainly for performing initialization tasks, are executed immediately after the process has been created. They do not have an input. Furthermore, they are executed before any other transition. However, all actions allowed for "normal" transitions are allowed for start transitions as well. Especially, setting variable values, setting timers and sending signals are the usual actions executed during a start transition. If no input from the environment is specified, the system is closed. In closed systems, at least one start transition has to send a signal or set a timer to ensure progress. Since all start transitions are executed before any of the other transitions in the system, their order is irrelevant. Any static order can be applied instead of executing all permutations. For example, assuming even a small system with only 10 processes, the number of transitions during reachability analysis is reduced from 10! (=3,628,800) to ten just for performing the start transitions.

This approach is already included in SFR-PO if the processes are located in different single fault regions. Here, it is extended to processes contained in the same single fault region.

5.3. Specification of Special Processes

In most cases, modeling a system is not limited to modeling the protocol specification. Auxiliary processes are often required to encapsule model-internal behavior easing the evaluation of the model. The most common model-internal processes are *Fault-Location* processes and *Evaluation* processes. Distinguishing these processes from processes implementing protocol behavior will also lead to a decreasing number of interleaved actions during reachability analysis, and thus to a reduction of the state space. This is discussed in the remainder of this section.

Fault-Location Process. The first step when validating fault-tolerance properties of a protocol is to select the faulty component(s). Instead of doing this manually for each set of components in turn, a process may be modeled to relief the user of this task. Within this fault-location process, a number of fault locations is specified. Different fault scenarios can be specified consisting of one or more fault locations, depending on the fault assumption. Upon start of the analysis, the fault location process selects the respective faulty processes for each scenario. It informs them about its decision by sending ordinary SDL signals. Afterwards, it terminates. Additionally, the fault type for each of the selected processes is determined and may be included as a parameter of the signal. Thus, the reaction of the system in the presence of faults occurring at different processes can be evaluated in a single validation run.

Such a process is not part of the fault-tolerance protocol model itself. It executes and terminates before the execution of the protocol is started. The resulting steps of performing an analysis are thus:

1. Initialization: Execution of start transitions.
2. Fault Location: Execution of the fault-location process' transitions.
3. Analysis: Execution of the protocol- model.

These steps can be executed one after the other without interactions. This yields an additional reduction of the state space due to less considered execution sequences.

Evaluation Process. After the fault-tolerance protocol has been executed, usually an evaluation is desired. In agreement protocols [EM96], for example, it should be checked whether all nodes have received the same value etc. This can be achieved by implementing an evaluation process that will start executing after the main fault-tolerance protocol (or one iteration of the protocol loop) has terminated. Like the fault-location process, it is not part of the protocol model, but is executed afterwards. Thus, the analysis sequence is further split into:

1. Initialization: Execution of start transitions.
2. Fault Location: Execution of the fault-location process' transitions.
3. Analysis: Execution of the protocol- model.
4. Evaluation: Execution of the evaluation process' transitions.

An alternative to the evaluation process are scripts or programs searching the reachability graph after execution of the analysis. The advantage of the evaluation process is that information can be gathered during execution of the protocol model thus the results of the analysis are available immediately after execution. Furthermore, the evaluation process is specified in the same language as the model.

5.4. Summary

In this chapter, different state space reduction techniques have been presented. A major contribution is the single fault region partial ordering (SFR-PO) mechanism. It shrinks the state space by evaluating only one representative execution sequence without skipping interesting behavior. The approach is based on the observation that (time-consuming) fault-tolerance mechanisms are located at the "entry-points" within the single fault regions. In other words, they check data incoming from other possibly faulty components. Thus, concurrency within the single fault regions is fully considered, but different permutations of interleaving actions between different single fault regions need not be considered − a single representative sequence is sufficient.

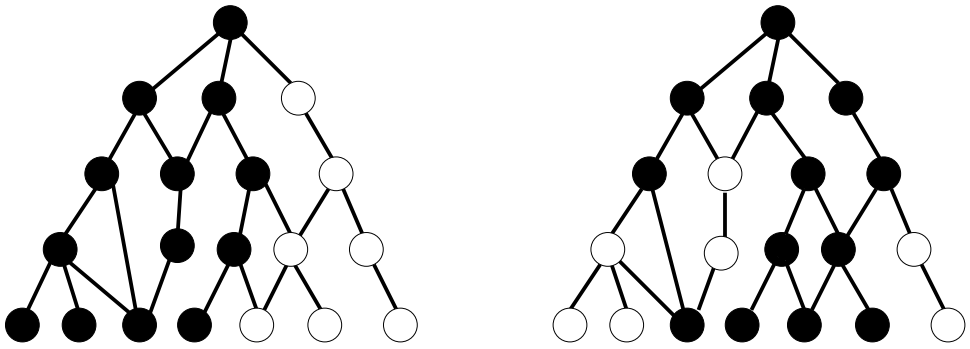
Further reduction techniques exploiting start transitions and definition of special processes for guiding the exploration have been introduced as well. These simple techniques lead to an additional reduction of the state space.

6. H-RAFT

So far, general state space reduction techniques for models of fault-tolerant protocols have been introduced. They can be combined with any algorithm, for example with the algorithms for reachability analysis in general as presented in section 3.2. While these algorithms perform well for rather small models with small state spaces, highly complex models with lots of arbitrary behavior cannot be investigated thoroughly. Despite the state space reduction techniques discussed throughout chapter 5, the resulting state space may still be too large to be explored completely. Thus, only a partial exploration can be achieved. The general algorithms of section 3.2 do not provide any means to guide the exploration. The two novel algorithms *H-RAFT* (*H*euristic *R*eachability *A*nalysis of *F*ault *T*olerant Protocols) as introduced throughout section 6 and *C2F* (*C*lose *TO* *F*ailure, section 7) provide heuristics to guide the (partial) exploration, thus increasing the chances for discovering fault-tolerance-property violations.

6.1. Introduction

Most of the algorithms for performing general (possibly exhaustive) reachability analysis are based on depth-first traversal. A weakness of pure depth-first algorithms is that they may provide a small dispersion only if the state space is too large to be explored completely. Figure 6.1 gives an exemplary illustration.



(a) Example for Rare Dispersion in Depth-first Traversal.

(b) Example Snapshot of a Reachability Graph Generated by H-RAFT.

Figure 6.1.: Comparison Between Depth-first Traversal and H-RAFT.

States that have been explored so far are represented by black circles. Unvisited states are depicted as white circles. Figure 6.1(a) gives an example of a straight-forward depth-first traversal. While ("the left") part of the reachability graph is explored in-depth, other parts may not even be touched within given time and memory limits.

The explored parts are likely to represent similar behavior or groups of behaviors. The later paths branch, the more common predecessor nodes they possess. Thus, the more likely that the remaining paths show similar results. Therefore purely depth-first oriented algorithms should be avoided.

Breadth-first algorithms guarantee optimal dispersion. Yet, a weakness of the fault-tolerance mechanism may be located deeply within the graph. Those depths might not be reached during a partial exploration. Thus, purely breadth-first based traversal is not suitable either.

The novel *H-RAFT* algorithm is similar to the exhaustive algorithm (section 3.2.1). Reconvergences will result in not exploring the path further. The partial exploration variant of the exhaustive algorithm through applying the bitstate method results in cutting paths arbitrarily. The criteria for further exploration of a path in *H-RAFT* is more sophisticated. It is based on a heuristic for transition selection through efficient off-line weight calculations. It allows for choosing the next transition from *all* global states explored so far. In the example of a typical H-RAFT-traversal in figure 6.1(b), transitions originating at the black circles − representing visited states − may be selected if they have not been executed yet.

Which transition should be chosen next? The perfect algorithm would chose the transition that continues a path leading to a fault-tolerance violation if there is any. Unfortunately, it is usually impossible to know a priori which path will lead to such a violation and whether such a path exists at all. By applying heuristics for selecting the next transition to be executed the probability of finding such a path should be increased. There are two general approaches on how to explore the state space:

1. Execute *all* active transitions of a global state. For the resulting global states, check which ones look "more suspicious" than the others. Continue with the states looking most suspicious and refrain from executing transitions of the other states and so on. The advantage of this approach is the solid base of information being available for taking a decision on whether to continue exploration from each state. For each generated state it can be easily checked whether a suspicion is justified. The drawback is the time and memory being "wasted" during run-time by executing transitions that are discarded afterwards.

2. Define a "suspicion-measure" (a weight, for example) for all active transitions. This measure could be based on criteria like "in this transition a lot of send operations are performed", or "a timer expires" etc. These criteria are run-time-independent. Thus, the weight of a transition can be computed off-line and is valid for *all* explorations of the model. Unlike in the first approach transitions with low weights are (possibly) not executed at all, thus saving time and memory. The drawback is that the decision on whether a transition is executed or not is based on less information than in the first approach as run-time dependent elements are not considered.

Table 6.1 summarizes the properties of the two approaches.

For the *H-RAFT* algorithm the two approaches are combined, thereby achieving a flexible selection method. Selection of the transition to be fired next is done in a two-step process:

Approach → **Property ↓**	**Approach 1**	**Approach 2**
suspicion detectable	YES	YES
run-time dependent	YES	NO
discard states after generation	YES	NO

Table 6.1.: Two Exploration Approaches.

1. A global state s_{next} of the already explored state space $stateSpace_{curr}$ is selected as the state the next transition will be chosen from. Selection is limited to those states containing enabled transitions.

2. The next transition tr_{next} is selected from the set of active transitions within s_{next}. The set of active transitions in any state s_i is termed $activeTRset(s_i)$. Active transitions of state s_i are all transitions that are able to fire in state s_i.

The advantage of this two-step selection method is that efficient off-line computation of transition weights can be combined with valuable run-time information. However, due to the pre-selection of a subset of global states in step one, it is not necessary to consider all active transitions of *all* global states. Thus, decision on tr_{next} is faster.
The heuristic for the first step selecting s_{next}, is introduced in section 6.2. Section 6.3 focuses on heuristics for choosing tr_{next} in the second step.

6.2. Global State Selection

The first step of selecting the transition that should fire next is to select one of the global states. Only those global states with active transitions are permitted for selection. Figure 6.2 gives an example: Only one of the black states surrounded by a grey ring may be chosen.

The set of global states with active transitions $stateSpace_{curr,act}$ is defined as

$$stateSpace_{curr,act} = \{s_i | s_i \in stateSpace_{curr} \wedge activeTRset(s_i) \neq \emptyset\}, \tag{6.1}$$

where $stateSpace_{curr}$ denotes the state space explored so far (called "current" state space, the black circles in figure 6.2) and the s_i represent the global states.

The set $activeTRset(s_i)$ comprises all active transitions of global state s_i. Active transitions are shown as black lines with grey shadow in figure 6.2. Note that there may be transitions from states in $stateSpace_{curr,act}$ that are not active as the event enabling those transitions has not been received.

Selection of the global state is based on weights. As global states emerge during analysis, their respective weight has to be determined during run-time. The overall global state weight may be composed of weights that can be determined off-line and of weights that are only available during run-time. The weight of global state s_i is denoted by $wState(s_i)$.

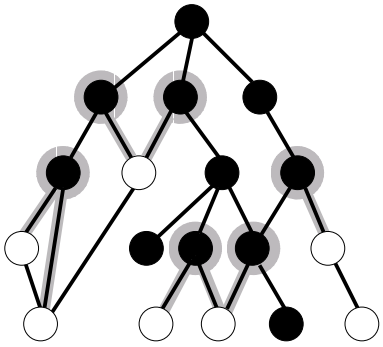


Figure 6.2.: Global State Selection.

Let $\mathcal{S}_{next}$ be the set of selectable global states with the highest weight:

$$\mathcal{S}_{next} = \{s_i | s_i \in \underset{s_i \in stateSpace_{curr,act}}{\operatorname{argmax}} wState(s_i)\}. \tag{6.2}$$

s_{next}, denoting the global state the next transition should be chosen from, is selected from $\mathcal{S}_{next}$ by applying a random selection among the candidates. In case only one element exists in the set, that global state is chosen.

So far, the general selection strategy for global states has been explained. Next, it is discussed how $wState(s_i)$ can be determined.

Basically, five criteria may be considered for inclusion in the global state weight calculation:

1. Transition weights;
2. Local main states;
3. Current model time;
4. User-defined criteria based on model knowledge;
5. Depth of the global state within the reachability graph.

Criteria 2 to 5 are run-time dependent, while transitions priorities can be computed off-line.

Transition Weights (Criterion 1). Transition weights should express the assumed chances of the transition being on a path leading to a fault-tolerance-mechanism violation. Thus, the weights of the transitions defined in global state s_i should be incorporated in the calculation of $wState(s_i)$. Transition weight calculation is done off-line and will be introduced in detail in section 6.3.

Equation 6.3 defines the global state weight $wState(s_i)$ for any global state s_i based on transition weights only:

$$wState(s_i) = \max_{tr_j \in activeTRset(s_i)} \{wTrans(tr_j)\}, \tag{6.3}$$

where $wTrans(tr_j)$ denotes the weight of transition tr_j (see section 6.3). Weight values are usually selected from $\mathbb{R}_0^+$, but they may be assigned values in the whole range of $\mathbb{R}$.
Equation 6.3 formalizes that the global state weight is set to the highest weight of its active transitions.

Other combinations of the active transitions' weights, for example their sum, to determine the global state weight would be possible as well. However, selection according to the maximum makes this criterion independent on the number of currently active transitions.

Local Main States (Criterion 2). Not changing local main states means a transition will (possibly) perform some actions and return to the main state. This may be an indicator, that the transition is not as "valuable" as a transition changing its main state after a transition. Entering new local main states could represent discovering new behavior. Thus, means should be provided to distinguish between these two transition types.

Every time a transition in one of the processes fires, a new global state is created representing the system state after transition execution. This is, of course only the case if no reconvergence is detected. If the new state is identical to an existing global state, a reconvergence has occurred and no new global state is created as the investigation of the path is aborted at that point. In case no reconvergence occurred, this leads to the observation that each global state will differ in (at most) one local main state from its predecessor in the reachability graph.

Transitions resulting in the same local main state after execution can already be considered in the transition weight (see section 6.3). The information of whether a different state is entered is available during off-line analysis. Thus, this criterion will not be explicitely applied to the global state weight calculation.

If variables are of importance, substates may also be considered. This extended criterion is not considered explicitly as changing a variable by an assignment is already included in the first criterion "Transition Weights".

Another criterion related to local main states is whether a local state has not been visited throughout the reachability graph explored so far. Although giving a higher weight to those transitions resulting in a so far unvisited state could be considered, this is rather a criterion for a coverage algorithm. Coverage of the static model states would be the goal there.

Current Model Time (Criterion 3). Current model time cannot be related to fault tolerance properties directly. One might argue that faults are more likely to happen at a later time during protocol execution. However, what is really meant is that faults are more likely to occur deeper in the reachability graph, as more behavior has been observed the deeper a state is located in the graph. Depth of the global state is another criterion and is discussed in a later paragraph.

Changes in model time from one global state to its successor, on the other hand, may represent typical behavior within a fault-tolerant protocol: Advancing model time may reflect the expiration of a timer, the use of computation time, the omission of an action due to a faulty component etc. Thus, these changes can be considered an important factor with respect to fault-tolerance protocols and thus should be included in the weight calculation.

This criterion can again be applied at transition level already and will be further discussed there (see section 6.3: transition input element "timestep"). Thus, it is not included in the global state weight explicitely.

User-defined Criteria (Criterion 4). User-defined criteria provide additional information that may be helpful for selection. Nevertheless, this factor will not be included into the weight calculation of the *H-RAFT* algorithm as it requires in-depth knowledge of the model. One of the goals of *H-RAFT* was to develop an algorithm that refrains from requiring knowledge of the model semantics. The *C2F* algorithm (see section 7) is mainly based on this criterion.

Depth of the Global State (Criterion 5). Complex model behavior may lead to deep state spaces, especially if the model is fine-grained. If dependencies to concurrent operations in the system exist, depth increases even more quickly. Fault-tolerance violations may thus be located deeply in the graph.

Algorithms exploring the state space depth-first will reach those depths, however, as already discussed in previous sections, pure depth-first traversal should not be applied. However, the current depth should be a factor for selecting global states. Thus, it can be guided whether and when deep parts are to be explored more thoroughly.

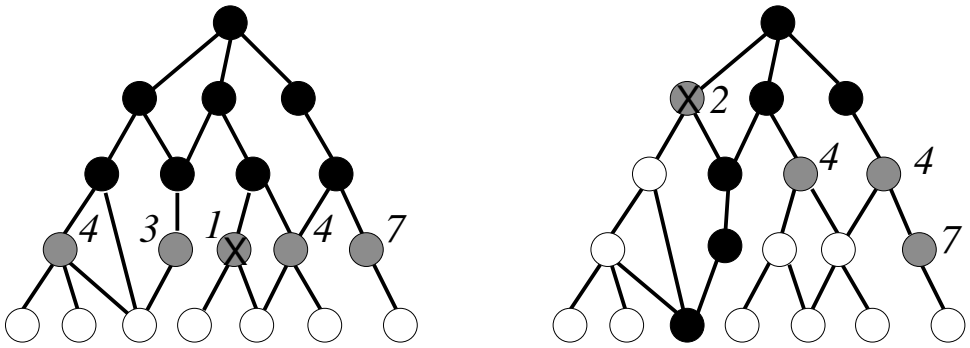
Ensuring minimum exploration depth could be accomplished by including a factor into the basic global state weight equation (6.3). However, this would result in a rather complex function for defining the factor. Instead an approach will be pursued to restrict the width of the reachability graph. Within given run-time and memory limits, only a certain number of states may be explored (partial exploration). Through restrictions in the width of the graph, more states located deeper within the graph will be reached while "important" parts are not skipped (as might be the case with pure depth-first traversal). Thus, depth can be guaranteed with minimum computation overhead as discussed throughout the following paragraphs.

Width Restriction. Width restriction of the reachability graph can be accomplished in two ways:

- Restriction on a *level* of the reachability graph.
- Restriction on the *front* of the reachability graph.

Level Restriction. An example for level restriction is shown in figure 6.3(a). The *level* of a global state s_i is defined by the distance of s_i from the root. The grey states in figure 6.3(a), for example, are on level 3. The root state is at level zero.

Black circles and grey circles indicate the global states that have already been generated. White circles show (currently) unexplored states. The difference between grey and black circles is for visualization of the level restriction idea only. They are all on the same level. In this example a width restriction of "4" is assumed. Thus, only 4 states per level may be considered for further exploration. If there are more global states on a level, there are 5 states on the grey level, the states with the lowest global weights are discarded. Their subtrees will not be investigated, except if there is a reconvergence leading to one of its successors. The numbers in figure 6.3(a) show the weight of each grey state. The grey state marked with a black X is the state that will not be considered any further as its weight of "1" is smaller than the weights of the other states on the respective level.



(a) Example for Level Restriction. (b) Example for Front Restriction.

Figure 6.3.: Examples for Width Restriction of the State Space; Allowed Width = 4.

Front Restriction. Figure 6.3(b) depicts an example for front restriction. Instead of limiting the number of global states per level, here the number of global states that may be considered at any point of time during the analysis is restricted. In other words the number of global states allowed in $stateSpace_{curr,act}$ is limited.

Grey circles in figure 6.3(b) indicate states on the current front. In other words: only those global states containing active transitions. Again, the states marked with a black X refer to states within the current front, that are excluded from further exploration due to the width restriction.

With front restriction the relevance of a global state with respect to fault-tolerance properties, i.e. its weight, is the only criterion for discarding or keeping it. Thus, this strategy can be considered more appropriate for detecting fault-tolerance violations than the level restriction as global states are not excluded just because they are on the "wrong level". As an example, assume the weights of all global states on a level x are in the interval $[a, b]$ and all global states on a level $x + \alpha$ have been assigned a weight of $[c, d]$, where $c < d < a < b$. Then level restriction would discard global states with high weights on level x and states with low weights on level $x + \alpha$. With front restriction, this can be avoided. All states on level x would be retained while more states of level $x + \alpha$ would be discarded.

Independent of whether level or front restriction is applied, the basic idea is always to discard the global states with the lowest weights from further exploration. This approach is justified if

a reasonable width is chosen such that it leaves still enough global states that may be selected as s_{next}.

Balance of Subtrees. Limiting the width of the reachability graph, by any of the two techniques, could result in an unbalanced exploration.

Usually, trees growing deeper will become wider. With a constant width restriction for all depths, the subtree growing from a single global state may, at some point, consume the whole allowed width. Thus, other subtrees may not be investigated as they are prevented from being explored. Figure 6.4 shows an example if level restriction is applied. The framed subtree outgrows the others. The same effect may occur with front restriction.

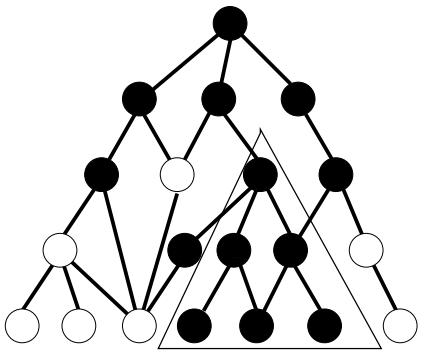


Figure 6.4.: Example for Outgrowing of a Single Subtree; Allowed Width = 3.

To prevent this outgrowing of a single subtree and to retain a more balanced exploration, the weight of states that are located deeper within the graph may be reduced. This measure is independent of whether a level- or a front-restriction is applied. It can be applied to both of the restriction criteria, additionally.

Combination of width restriction and reduction of the weight of deeper global states guarantees on the one hand that deep states may be reached, and on the other hand that the traversal is balanced to some extent. Thus, the improved equation 6.4 for $wState(s_i)$ includes a depth reduction factor $df(depth(s_i))$ based on the depth $depth(s_i)$ of s_i within the reachability graph. Equation 6.4 is derived from equation 6.3.

$$wState(s_i) = \max_{tr_j \in activeTRset(s_i)} \{wTrans(tr_j)\} - df(depth(s_i)). \tag{6.4}$$

$df(depth(s_i))$ may be assigned values in $\mathbb{R}$, however values in $\mathbb{R}_0^+$ should be preferred. For $df(depth(s_i)) = 0$ equation 6.3 is obtained.

Selection of a suitable df-function should rather be defined relative to the overall depth of the state space. However, this depth is not known in advance, thus alternatives have to be sought. It has been refrained from a multiplicative combination in order to allow for more flexibility of defining the impact of the depth factor on the overall weight. Equation 6.4 may also yield

negative weights depending on the definition of df. This is not a drawback, but increases diversity of the weights.

Different depth functions df in conjunction with different allowed widths will be evaluated in section 12.1.

6.3. Transition Selection

In the previous section the method to select the next global state s_{next} has been described. Furthermore, a way to ensure depth and somehow balanced exploration has been presented. This section describes how the next transition to be fired (tr_{next}) is selected.

Selection of tr_{next} is similar to the selection of s_{next}. It is the active transition tr_j with the maximum weight $max\{wTrans(tr_j)\}$ in the set of active transitions $activeTRset(s_{next})$ of s_{next}.

Equation 6.5 formalizes the set $\mathcal{T}_{next}$ of transitions that may be selected. Again, a random selection is applied if there is more than one element in $\mathcal{T}_{next}$ and thus tr_{next} is not unique. *Argmax* refers to the argument (here: tr_j) where the equation evaluates to the highest value.

$$\mathcal{T}_{next} = \{tr_j | tr_j \in \underset{tr_j \in activeTRset(s_{next})}{\operatorname{argmax}} wTrans(tr_j)\}, \tag{6.5}$$

where $wTrans(tr_j)$ denotes the weight of transition tr_j.

Transition Weight Calculation. Transition weight calculation of $wTrans(tr_j)$ for each transition tr_j, is based on the language elements of SDL. This allows for efficient off-line transition weight calculation. This is one of the main goals designing *H-RAFT*.

On-line weight calculation would allow for inclusion of run-time dependent factors like current variable assignments etc. Consideration of such factors may yield improvements as more information for transition selection is available. In *H-RAFT*, changes of variable assignments are considered (see section 6.3.1), but not the actual contents of the variables. Through this restriction, on-line calculation can be avoided.

SDL language elements can be subdivided into two groups (see also section 2):

- transition input elements and
- transition action elements.

Transition input elements comprise signal consumption, timer expiration/consumption, spontaneous transition execution and model time progress. Model time progress is an implicit input element. In other words: every event triggering a transition tr_j. Transition input elements and their contribution to the overall transition weight are discussed in subsection 6.3.1.

Transition action elements subsume the actions performed during transition execution, like sending signals, setting/resetting timers, assigning variable values, comparisons etc. They are presented in detail in section 6.3.2 and their inclusion in the overall transition weight is discussed.

6.3.1. Input Weights

Let $input(tr_j)$ denote the input element (= input event) of transition tr_j. There are the five explicit input elements as described in section 2.3.2 and two implicit ones. All seven elements are described (implicit elements) or shortly recalled (explicit elements) in the following list. Furthermore, their correlations to models of fault-tolerance protocols are discussed.

- $input(tr_j)$ = ***timer expired***: Transition tr_j is enabled because of an expired timer.

  Timers are a very important means in modeling fault-tolerant protocols. Their expiration may express the end of deadlines. Missing deadlines may be an indicator for a fault occurrence. Which in turn could result in a violation of the fault-tolerance properties.

- $input(tr_j)$ = ***timer array expired***: Transition tr_j is enabled by an expired timer that is part of an SDL timer array.

  Modeling timers as arrays may be an indicator that the process containing the timer array is a delay process (see section 5.1.2). This is dependent on the modeling style. However, as timer arrays are highly suitable for modeling delay processes it can be assumed that most modelers follow this style. Differentiating between "normal" timers and timer arrays allows for distinguishing those processes from other ones. Timer array expiration does not yield a missed deadline - at least not as a direct consequence. Thus timer arrays can be assumed to be less critical for finding a fault-tolerance violation than "normal" timers.

- $input(tr_j)$ = ***none***: Transition tr_j is enabled spontaneously.

  Spontaneous transitions in conjunction with faulty processes are discussed in paragraph "*Spontaneous Transitions in Faulty Processes*" later in this section.

  In fault-free processes, spontaneous transitions may be used to specify that an event has occurred within an interval. While the upper bound, possibly representing a deadline, is defined by a timer, firing of a spontaneous transition before that timer elapses may indicate that the event has happened on time. Thus the input element *none* can be seen as counterpart to "normal" timers and is therefore an important element in fault-tolerant models.

- $input(tr_j)$ = ***signal with parameters***: Transition tr_j is enabled by consumption of a signal containing parameters.

  Parameters reflect information flow from one component to another. This includes reception of corrupted values from faulty nodes which, in turn, may lead to those model parts representing the fault-tolerance mechanism.

- $input(tr_j)$ = ***signal without parameters***: Transition tr_j is enabled by consumption of a signal not containing any parameters.

  Signals without parameters are often used as model-internal communication means. Thus, they should be distinguished from signals containing parameters.

- $input(tr_j)$ = ***timestep***: If no transition − with the exception of possibly defined spontaneous transitions − is active, model time may advance. If model time advances while spontaneous transitions are enabled, this represents that the spontaneous transitions did

not fire. The transition input element *timestep* refers to performing the time progress. It is thus only an implicit SDL input element. It causes generation of a new global state just as "real" transitions do. Transition from level 4 to level 5 in figure 3.3 on page 25 gives an example for a *timestep*-transition.

With respect to the fault-tolerance protocols, this input element represents the counter-weight to spontaneous transitions: The higher this weight, the more likely a spontaneous transition will not fire, but model time advances.

- $input(tr_j) =$ ***timer ready***: This implicit SDL element expresses that a timer has expired and is being inserted into the input queue of the receiving SDL process. Thus, it enables transitions depending on the respective timer. It precedes every "timer expired" element.

  The relevance of this input element is ambiguous. It can be argued that it is covered by the *timer expired* element, however, it may also be that a timer expires, is inserted in the input queue and then discarded. Thus, the *timer expired* event will not occur. This happens if there is no transition defined depending on the timer as input element within the current local state of the process. Thus, the two elements should be considered separately.

In order to be able to consider all of the seven input elements in the overall transition weight calculation, each of the elements is assigned a static weight:

$$wInput(input(tr_j)) = \begin{cases} wTimerExp, & \text{if } input(tr_j)\text{="timer expired";} \\ wTimerArrayExp, & \text{if } input(tr_j)\text{="timer array expired";} \\ wNone, & \text{if } input(tr_j)\text{="none";} \\ wSignalWithParams, & \text{if } input(tr_j)\text{="signal with parameters";} \\ wSignalWithoutParams, & \text{if } input(tr_j)\text{="signal without parameters";} \\ wTimestep, & \text{if } input(tr_j)\text{="timestep";} \\ wTimerReady, & \text{if } input(tr_j)\text{="timer ready".} \end{cases}$$

As each transition contains exactly one input element, the overall weight $wTrans(tr_j)$ can be defined by the static transition input weight of the corresponding transition tr_j:

$$wTrans(tr_j) = wInput(input(tr_j)). \tag{6.6}$$

This equation will be extended with the introduction of transaction action elements in section 6.3.2.

Spontaneous Transitions in Processes Representing Faulty Components. In equation 6.6, the static weights are not modified during transition weight calculation. In this paragraph the only exception is discussed: The weight for spontaneous transitions in faulty processes may be modified during the analysis.

When investigating fault-tolerance properties of a protocol, faults need to be modeled as well. In [Kes02] implicit modeling of faulty behavior in SDL has been discussed. The most universal fault-model "any output at any time" comprises arbitrary faulty behavior in the value and time domain. By applying this model, any kind of faulty behavior is investigated.

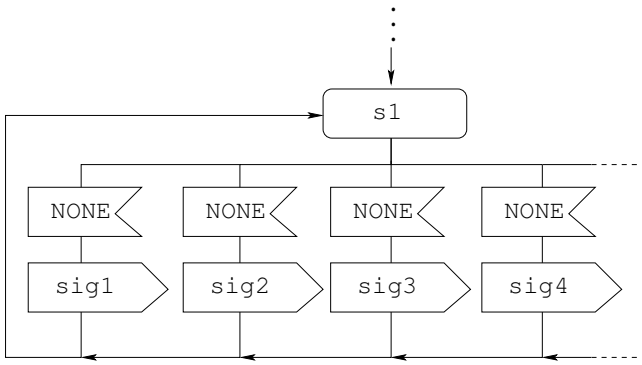


Figure 6.5.: Example Model of A Faulty Component Exhibiting "any output at any time" Behavior.

- Omission faults are covered as spontaneous transitions do not necessarily have to fire.
- Timing faults in general are covered as spontaneous transitions may fire at any time.
- Faults in the value domain are achieved by specifying multiple spontaneous transitions in the same local state of the process representing (part of) a faulty component; each of these transitions sets different variable values.

Modeling faults in SDL according to the "any output at any time" paradigm can be accomplished comfortably through (possibly many) spontaneous transitions.

A small example of a process modeling a faulty component with faulty behavior according to "any output at any time" is depicted in figure 6.5.

Spontaneous transitions may not only fire at any time, but also repeatedly. In the typical example in figure 6.5, each transition results in the same state, therefore the spontaneous transitions are enabled immediately again. Thus, the state space is possibly growing rapidly. This state space explosion has to be prevented.

A solution to this problem is to restrict the number of, otherwise arbitrary, repetitions of spontaneous transitions in those processes. In the remainder of this paragraph a mechanism is provided to restrict the number of times each spontaneous transition within a process, representing a faulty component, may fire on each path of the reachability graph.

This approach does not limit the points in time when those transitions may fire, just the number of times they may fire and covers most of the possible faulty behavior. Excluding redundant repetitions is crucial for state space reduction. Otherwise, the number of (supposedly unnecessary) repetitions may grow too large. Executing actions leading to a fault (like sending the same wrong signal) multiple times rarely creates new faulty behavior. This is the case in many well-known fault-tolerance techniques where only a limited number of repetitions is of interest. This behavior has also been observed in extensive experiments with large models.

Conclusion: This limiting technique is a suitable compromise.

Let *nDec* denote a "decrease factor". *nDec* is used to transform the static weight for spontaneous transitions *wNone* into a dynamic weight iff the spontaneous transition is located in a process representing (parts of) a faulty component. Let $\mathit{fired}(tr_j)$ count the number of times transition tr_j has fired on the current path in the reachability graph. Then, the dynamic weight for spontaneous transitions in processes expressing faulty behavior is defined by equation 6.7.

$$wInput(input(tr_j)) = wNone \cdot nDec^{\mathit{fired}(tr_j)}, \tag{6.7}$$

iff $input(tr_j) = "none_{faulty\_process}"$.

A multiplication has been selected in order to ensure the dynamic weight will quickly approach zero. In case of subtraction from *wNone*, a basic weight would be retained nevertheless. This could result in the transition being still fired very often. Through multiplication this is prevented.

For $nDec \in [0, 1]$ equation 6.7 indicates that every time transition tr_j fired on the current path, its weight is decreased exponentially in subsequent activations along the path. A decrease factor of zero indicates a constant weight of zero for all spontaneous transitions in processes implementing faulty behavior at all times, while a decrease factor of 1 implies a constant weight equal to the weight *wNone* of spontaneous transitions in processes modeling fault-free components.

Signal-Consumption-Only Transitions. Equation 6.6 is applied for calculation of all transition weights. The modification presented in the previous paragraph only extended a static weight assignment to a dynamic one. It did not restrict the applicability of equation 6.6.

This paragraph introduces the only exception: *Signal-Consumption-Only* transitions (short: *SCOs*). SCOs subsume transitions that consume their input signal, but neither change their local state (they result in the same local state), nor perform any action during the transition. Thus, they do not provide substantial progress during reachability analysis as signal consumption is their only contribution. These transitions can be identified easily during off-line analysis.

Signal-Consumption-Only transitions are set to a static weight as well:

$$w(SCO) = wSCO. \tag{6.8}$$

$w(SCO)$ is independent of the transition input element. Thus, equation 6.6 has to be extended to:

$$\mathcal{T}_{next} = \{tr_j | tr_j \in \underset{tr_j \in activeTRset(s_{next})}{\operatorname{argmax}} \begin{cases} w(SCO), & \text{if } tr_j \in \mathcal{SCO} \\ wTrans(tr_j), & \text{else} \end{cases} \}, \tag{6.9}$$

where $\mathcal{SCO}$ is the set containing all Signal-Consumption-Only transitions.

Appropriate values for $w(SCO)$ are discussed throughout sections 12.1 and 12.2.

6.3.2. Action Weights

So far, transition input elements and their weights have been introduced (section 6.3.1). Their relation to fault-tolerance protocols has been pointed out. The resulting weight function for transitions is simple so far, as there is only one input element per transition.

Handling of transition action elements is not as straightforward as there may be more than one action per transition. Thus, not only the static weights have to be considered. Their combination with each other has to be taken into account. Furthermore, combinations with the transition input element weights have to be discussed.

First, the transition action elements and their relations to fault-tolerance protocols are presented. Then weight-combination possibilities are introduced culminating in the final definition of transition weights.

Transition Action Elements. The following list summarizes all action elements and discusses their relation to fault-tolerant protocols.

- ***Sending signals.*** Each signal that is sent may result in a transition being enabled at another process. This is, if the receiving process has a transition with that signal as input event to one of its transitions. Otherwise, the signal is discarded. If it's not discarded, more transitions are available for selection in the next round. Thus making the decision more reliable as more choices are present.

  Moreover, (multiple) sending operations in processes modeling faulty behavior may represent a source of Byzantine faults. Different signals may be sent to adjacent processes.

  In contrast to the input elements, for this action element it is not distinguished between signals carrying parameters and those without parameters. The number of enabled transitions is increased in both cases.

- ***Setting timers.*** The importance of timers has already been discussed for the transition input elements. Setting a timer either results in an expiration of that timer or in a reset. Thus, setting timers does not always result in the *timer ready* input event. Therefore, this parameter should be distinguished from the *timer ready* input event.

- ***Setting timers in timer arrays.*** The action of setting a timer in an array of timers can be related to fault tolerance properties as setting a "normal" timer. Additionally, it can be argued as for the corresponding input element *timer array expired*. As timer arrays are likely to appear in delay processes, it is more unlikely, that any of those timers is reset. These timers then do not represent deadlines. Thus, distinguishing between setting "normal" timers and timers in arrays is necessary.

- ***Resetting timers.*** This action element could express that some event has happened before a deadline has expired. Therefore, resetting timers could be an indicator that no fault has occurred, or a fault has been tolerated.

  It is not distinguished between resetting "normal" timers and resetting timers in arrays, as the latter one is rarely expected to happen.

- ***Modification of variables.*** Changing contents of variables may represent manipulation of counters, possibly counting missed or corrupted messages. Furthermore, this action is the only one capturing changes in the value domain. Modification of a variable may also mean, that it is set to a faulty value. Those faulty values may also represent memory faults that have been included in the model.

Action Weight Calculation. For transition input elements, the respective input weight could be used as transition weight directly as there is exactly one input event to every transition. As a transition usually contains more than one action, this strategy cannot be used for transition action elements. In this paragraph, different strategies of combining the multiple action weights contained in a transition to an overall action-transition weight are discussed. In section 6.3.3, approaches on how to combine input and action weights into one equation for the overall transition weight are introduced.

As for the input weights, each of the five action elements is assigned a static weight. In other words: each occurrence of the respective action element is assigned the same weight. Multiple occurrences and combinations of elements are discussed in subsequent paragraphs. The static weights will be referred to as action weights and are defined as follows:

$$wAction(action) = \begin{cases} wSendSig, & \text{if } action\text{=”sending signals”;} \\ wTimerSet, & \text{if } action\text{=”setting timers”;} \\ wTimerArraySet, & \text{if } action\text{=”setting timers in timer arrays”;} \\ wTimerReset, & \text{if } action\text{=”resetting timers”;} \\ wVarChange, & \text{if } action\text{=”modification of variables”.} \end{cases}$$

Let $action(tr_{j,k})$ denote the k^{th} action element in transition tr_j. Then the weight of action element k within transition tr_j can be expressed by $wAction(action(tr_{j,k}))$. The total action weight $wAction(tr_j)$ of transition tr_j can thus be defined as a function over all transition action elements:

$$wAction(tr_j) = f_{action}(\mathcal{A}_j), \tag{6.10}$$

The set of all action elements contained in transition tr_j will be denoted by $\mathcal{A}_j$.

The following list contains examples for defining function $f_{action}(tr_j)$ if the action elements are considered to be independent of each other:

- $f1_{action}(tr_j) = \max_{k \in \mathcal{A}_{UNIQUE}} \{wAction(action(tr_{j,k}))\}$.

  The highest static weight of the action elements occurring in tr_j is selected.

- $f2_{action}(tr_j) = \sum_{k \in \mathcal{A}_{UNIQUE}} wAction(action(tr_{j,k}))$,

  where set $\mathcal{A}_{UNIQUE} \subset \mathcal{A}$ contains only one instance of each of the action elements occurring in tr_j. Thus, $|\mathcal{A}_{UNIQUE}| \leq 5$.

- $f3_{action}(tr_j) = \sum_{k \in \mathcal{A}} wAction(action(tr_{j,k}))$.

  The weights of all k action elements constituting transition tr_j are added up.

$f1_{action}(tr_j)$ represents the most simple of the three functions. Its disadvantage is that it can result in at most five different values: As there are five action elements, five static weights are available. Thus, applying this function may lead to a nearly random algorithm assuming a uniformly distributed selection is applied among transitions with the same weight.

$f2_{action}(tr_j)$ proposes a slightly better dissemination. However, if the same types of action elements appear in many transitions, the dissemination would still be close to the one of $f1_{action}(tr_j)$.

$f3_{action}(tr_j)$ is the most fine-grained function of the three functions. It yields the best dissemination as both the different types of action elements and the number of their respective occurrences is considered. Thus, this function should be preferred.

With respect to computation overhead no drawback is implied by the more complex functions as all calculations can be performed offline.

More complex functions $f_{action}(tr_j)$ have to be defined if the action elements are not considered independent of each other. There may be representative combinations of some of the action elements within a transition tr_j. For example, resetting a timer and sending at least one signal could be an indicator of an event occurring on time etc. These combinations, or transitions containing these combinations could be assigned a combined weight. More generally: There may be combinations of action elements that suggest typical transitions or processes in models of fault-tolerance protocols. Weights for such transitions or even for complete processes containing one or multiple of those transitions can be assigned a different weight (possibly) independent of the static transition action element weights. Usually, these transitions cannot be identified without their input element. Thus, examples of typical transitions of fault-tolerant protocols will be presented in 6.3.3 in conjunction with the discussion on how to combine input and action element weights into an overall transition weight.

6.3.3. Transition Weight Composition

So far, transition weight calculation based on transition input elements only (equation 6.6) and calculation of a transitions overall action weight, based on transition action elements only (equation 6.10), have been introduced. In this section, the equation for overall transition weight calculation (so far equation 6.6) is extended to include action weights. Furthermore, strategies on the selection of the input and action weights relative to each other are discussed.

The overall weight formula for transition tr_j including action weights is extended to equation 6.11.

$$wTrans(tr_j) = \alpha \cdot wInput(input(tr_j)) + \beta \cdot wAction(tr_j). \tag{6.11}$$

The factors α and β allow for biasing, or balancing the two weights. The following discussion is based on the assumption that the static weights of both, the input and action elements,

are chosen in the same order of magnitude. Selection of α and β can be based on different preferences:

- The input element is considered the most important element of the transition:

  In this case, the action weights are merely used as minor factor of the overall transition weight. Its main function is to give preferences to the transitions with equal input weights. Thus, the random selection among those candidates with equal (input) weights does not have to be applied. Let *maxStaticWeight* denote the highest static weight that may be assigned to any input and action element. With the static input and action weights being in the same order of magnitude, a possible assignment for α and β may be:

$$\begin{aligned} \alpha &= \mathit{maxStaticWeight}. \\ \beta &= \begin{cases} \frac{1}{|\mathcal{A}|}, & \text{if } \mathit{f3}_{action}(tr_j) \text{ is used;} \\ \frac{1}{|\mathcal{A}_{UNIQUE}|}, & \text{if } \mathit{f2}_{action}(tr_j) \text{ is used;} \\ 1, & \text{if } \mathit{f1}_{action}(tr_j) \text{ is used;} \\ 0, & \text{if the action weights should not be considered at all.} \end{cases} \end{aligned}$$

- The action elements are considered more important than the input element:

  In this case, the assignment for α and β is independent of the three functions $f_{action}(tr_j)$:

$$\begin{aligned} \alpha &= \begin{cases} 0, & \text{if the input weight should not be considered at all;} \\ 1, & \text{else.} \end{cases} \\ \beta &= \begin{cases} 1, & \text{if } \alpha = 0; \\ \mathit{maxStaticWeight}, & \text{else.} \end{cases} \end{aligned}$$

- The input weight and the overall action weight should be considered equally, i.e. $\sum$ input $\approx \sum$ action:

  Selection of α and β may be as follows:

$$\begin{aligned} \alpha &= 1. \\ \beta &= \begin{cases} \frac{1}{|\mathcal{A}|}, & \text{if } \mathit{f3}_{action}(tr_j) \text{ is used;} \\ \frac{1}{|\mathcal{A}_{UNIQUE}|}, & \text{if } \mathit{f2}_{action}(tr_j) \text{ is used;} \\ 1, & \text{if } \mathit{f1}_{action}(tr_j) \text{ is used.} \end{cases} \end{aligned}$$

- No difference in the importance between input and action elements is made, i.e. "single input weight" $\approx$ "single action weight":

  Thus, the assignment for α and β is simply:

$$\begin{aligned} \alpha &= 1. \\ \beta &= 1. \end{aligned}$$

The appropriateness of each of the four settings is investigated throughout section 12.3.

Typical transitions of fault-tolerant protocols. So far, overall transition weight calculations have been introduced based on the assumption that all input and action elements are independent of each other. The remainder of this section is dedicated to discussing possible dependencies. Certain combinations of input and action elements within a transition may be used to classify a transition or even a whole process with respect to its role in the model of the fault-tolerance protocol. This classification could be used to assign an overall transition weight to those transitions − or even to all transitions within the process − depending on their suspected importance with respect to fault tolerance. This assignment may be based on the constituting elements, for example modifying the overall transition weight − that would be computed if the elements were considered to be independent of each other − by a predefined factor. It may also be (almost) independent of the constituting elements: The elements are only required for determining the type of transition, but the static weights are ignored.

The following list contains transitions typically found in models of fault-tolerance protocols and the transition elements that allow for determining their type.

- **Incoming Signal in Delay Process:** This transition type models the arrival of a signal from one process. This signal has to be delayed for some time and is then forwarded to another process. Characterizing elements are:
  Input : *signal with parameters* representing the signal that should be delayed.
  Actions : *setting timers in timer arrays* to define the delay duration;
  modification of variables to increase the counter for next free position in the timer array;
  modification of variables to store the signal;
  nextstate - returning to the same local state.
- **Forwarding Signal from Delay Process:** This transition type models the expiration of a delay duration within a delay process. The signal is forwarded to the receiving process. Characterizing elements are:
  Input : *timer array expired* the timer associated with the signal that should be sent.
  Actions : *sending signals* forwarding the respective signal;
  nextstate - returning to the same local state.
- **Signal Arrival On Time:** Signals arriving on time at a process are modeled by this type of transition. Characterizing elements are:
  Input : *signal with parameters* the arriving signal.
  Actions : *resetting timers* the timer indicating the deadline for this signal is reset.
- **Sending Signal On Time:** This transition type is the corresponding type to *Signal Arrival On Time* at the sender side. Characterizing elements are:
  Input : *none* indicating the sending point before a timer expired.
  Actions : *resetting timers* the timer indicating the deadline for sending this signal is reset;
  sending signals for sending the signal.

These special transitions are grouped in a set $\mathcal{ST}$, for better reference. The weight for each of the special transitions is denoted by $wSpecial(tr_j), tr_j \in \mathcal{ST}$.

Equation 6.11 has to be extended to its final version:

$$wTrans(tr_j) = \begin{cases} wSpecial(tr_j), & \text{if } tr_j \in \mathcal{ST}; \\ \alpha \cdot wInput(input(tr_j)) + \beta \cdot wAction(tr_j), & \text{else.} \end{cases} \quad (6.12)$$

wSpecial(tr_j) for `Signal Arrival On Time` and `Sending Signal On Time` should be assigned a high value. These transitions are usually important for the fault-tolerance mechanism. Transitions handling signals in conjunction with delay processes represent properties of the physical layer, not of the fault-tolerance protocol itself. Thus, a small weight is preferable. Different weights for *wSpecial(tr_j)* are discussed in section 12.4.

6.4. Summary

The H-RAFT algorithm has been presented throughout chapter 6. The heuristic behind this algorithm is based on a two-step transition selection strategy. First, a global state is selected and the exploration continues at that state. This selection is based on available run-time information and ensures a traversal of the reachability graph in both (width and depth) directions. Width restriction and balancing of subtrees are the strategies employed there.

Transition selection is done in the second step. Each transition is assigned a weight that can be calculated off-line. It is distinguished between "normal" transitions and special transitions. The weight of a "normal" transition is calculated by combining the static weights of its input element and its action elements. Special transitions refer to transitions that can be distinguished to fulfill certain tasks typical to fault-tolerance mechanisms − for example transitions constituting delay processes. These transitions can be assigned to have a static weight independent of its elements. Equation 6.12 formalizes the overall transition weight calculation.

An evaluation of the H-RAFT algorithm, with different parameters and static weights, is provided in chapter 12.

7. Close-to-Failure

In this chapter, a second algorithm termed Close-to-Failure (C2F) is introduced. With the H-RAFT algorithm, no knowledge of the model is required by the user. It is straightforward to assume that including existing model knowledge improves the chances for finding violations of the fault-tolerance properties. The model knowledge has to be related to the fault-tolerance mechanisms. In other words: If it is known that certain properties during model exploration indicate that the analysis is steering towards a fault or a critical part in the protocol, that part of the state space should be explored in detail. More precisely, the user-provided information should represent an indication on how close to a fault-tolerance violation the exploration is.

Thus, a "distance measure" has to be defined. In [EN99] switches have been defined for indicating the distance to danger in safety-critical systems. The approach of the Close-to-Failure algorithm is roughly based on the same idea. The user may specify properties over global states and paths of global states. Furthermore, he may assign different relevances on how important each property is. In section 7.1, this criteria-defined approach is introduced. Section 7.2 includes some variants of the property definitions. The chapter is completed by a discussion on how to combine the Close-to-Failure approach with H-RAFT in section 7.3 and a summary in section 7.4. An evaluation of C2F is provided in chapter 13. It is compared to the other algorithms in section 14.

7.1. Criteria Definition

When investigating fault-tolerance mechanisms of protocols, properties with respect to those mechanisms have to be specified. These properties may either indicate a violation or the correct behavior of the protocol. As property violations are usually harder to specify − they may be unknown in advance − properties that indicate correct behavior of a protocol are preferable. A common property for agreement protocols, for example, is that after protocol execution all fault-free receivers agree on the same value. This is a single, simple property, especially if an evaluation process (see section 5.3, page 54) is implemented. If comparison etc. is already performed in such a process it may be sufficient to check whether the evaluation process reaches a state indicating correct behavior. Even in the absence of such a process, the property is easily defined: For each process representing a fault-free receiver the values have to be equal if all processes are in their final state. In other words: all nodes finished protocol execution and the values are equal. This property will be termed *final rule*, denoted by f in the following. f has to be evaluated after investigation of each new global state. It evaluates to either `true` or `false`.

The rule may be specified over a single global state or a path of global states − independent of the protocol type. The example of the previous paragraph is defined over a single global state: all processes have to be in certain local states and the values have to be equal within a single global state. If changes of variables or sequences of signal arrivals etc. are part of the rule,

paths of global states have to be observed. For example a property of the type "if action a has occurred, action b must eventually occur". Generally, the following spectrum of temporal logic may be used for rule specification:

$\forall \Box$ `property`	:	`property` holds at all times;
$\forall \Diamond$ `property`	:	on each path `property` is fulfilled at some time;
$\exists \Box$ `property`	:	on at least one path `property` holds at all times;
$\exists \Diamond$ `property`	:	on at least one path `property` is fulfilled at some time;
`property1` $\rightarrow$ `property2`	:	if `property1` has been fulfilled on one path, `property2` will finally also be fulfilled along that path.

Failure Property Definition. Specification of "critical" properties can be accomplished through a mechanism similar to the specification described above. Instead of defining a single final rule f, the user may specify a set of rules, denoted by $\mathcal{E}$ further on. The rules e_i within set $\mathcal{E}$ represent properties the user considers "critical" or relevant with respect to the fault-tolerance mechanisms of the modeled protocol. The rules e_i differ from f in two ways: First of all, f expresses a *safe* state while the e_i express *critical* states. The second difference is that the e_i do not evaluate to a Boolean result. They should express the relevance of a property with respect to the fault-tolerance mechanisms. Thus, the results will be in the interval $[0, 1]$. For each rule, the user may assign a different relevance (= weight) within this range. As an example the following rules could be specified for the signed messages protocol (see section 10.2):

$\mathbf{e_1}$(at least two signatures are equal) = 0.6;

$\mathbf{e_2}$(at least one signature is invalid) = 0.3;

$\mathbf{e_3}$(a value in the consistency vector is changed) = 0.1.

In this example the user considers equal signatures more critical than invalid signatures or variable changes.

Global State Selection. Evaluation of the e_i is performed each time a new global state has been investigated. The results $e_i(s_j)$ for all rules in $\mathcal{E}$ are attached to the respective global state s_j.

Following the two-step selection approach as introduced in H-RAFT, the global state with the maximum weight result of all active states ($stateSpace_{curr,act}$, see page 57) is selected. For this selection, an overall global state weight has to be determined. In other words: a function e has to defined over the e_i combining their weights into a single one. Two combinations are straightforward:

MAXIMUM : $e_{MAX}(s_j) = \max_{e_i \in \mathcal{E}} \{e_i(s_j)\}$

AVERAGE : $e_{AVG}(s_j) = \sum_{i=1}^{|\mathcal{E}|} e_i(s_j) \ / \ |\mathcal{E}|$

$e_{MAX}(s_j)$ yields the highest value of the currently fulfilled properties within state s_j. Exploration is continued with the global state (within $stateSpace_{curr,act}$) with the highest e_{MAX}. The advantage is that the currently highest relevance is considered.

$e_{AVG}(s_j)$ represents the average over all e_i results. The advantage of this weight combination is that not only one property determines the decision. The more properties are fulfilled the higher the chances a transition is selected.

Table 7.1 contains an illustrative example for the two selection methods:

i →	**1**	**2**	**3**	**4**	**5**
$e_1(s_i)$	0.1		0.1	0.1	
$e_2(s_i)$	0.3	0.3			
$e_3(s_i)$	0.2		0.2	0.2	
$e_4(s_i)$					0.8
$e_5(s_i)$	0.4		0.4		
$e_{MAX}(s_i)$	0.4	0.3	0.4	0.2	**0.8**
$e_{AVG}(s_i)$	**0.2**	0.06	0.14	0.06	0.16

Table 7.1.: Example for Global State Weight Calculation in C2F.

In the example, five rules e_1 to e_5 are considered. The columns represent the currently active states s_1 to s_5. In each cell, the relevance of the respective property is displayed if the property is assumed fulfilled. If not fulfilled, the cell remains empty. This represents a value of zero indicating the rule is assumed not critical with respect to fault-tolerance properties. The last two rows show the calculated overall global state weights when applying e_{MAX} and e_{AVG}. With e_{MAX}, global state s_4 would be selected. Application of e_{AVG} would result in choosing global state s_1.

In table 7.1, the resulting weights for global states s_1 and s_3 (resp. s_2 and s_4) are equal. In the example, this is irrelevant as neither of the states would be selected. However, it may occur that two or more global states with identical highest weight are available for selection. In that case a strategy has to be applied to decide on one of those states. Instead of randomly choosing a candidate, the next lower relevance could be considered for the e_{MAX} function. Thus, s_1 would be preferred over s_3 as e_2 is fulfilled in s_1. For the e_{AVG} function, the highest relevance (i.e. e_{MAX}) could be considered preferring s_2 over s_4.

An experimental evaluation of the global state weight calculation strategies is provided in chapter 13.

Transition Selection. After selection of a global state, an active transition of that global state has to be selected. Let $activeTRset(s_i)$, again, represent the set of transitions that are able to fire in global state s_i. In H-RAFT, transition selection was based on the static elements constituting a transition. For the Close-To-Failure approach, this idea is adopted in so far as transition elements will form the basis for the transition weights.

In H-RAFT each occurrence of an element has been assigned the same weight. This concept is extended for C2F: weight assignment is no longer based on the type of action, but the variable,

timer or signal that is part of the element. Variables, timers and signals will be referred to as items further on. Each item may be assigned a different user-defined weight. For example, if the user considers variable *var1* "critical" with respect to the fault-tolerance mechanism, he may assign a high weight to item *var1*. Weights of items are denoted by *w(item)*. If the user does not assign a specific weight to an item, a default of zero is assumed. The transition weight calculation is thus again only dependent on the static model and can be computed off-line. Run-time dependent changes, for example assignment of a specific value to a variable, are already considered in the global state weight calculation. Such assignments are included in the e_i.

Through this method of weight calculation of both, global state weights and transition weights, the two-step selection method as used in H-RAFT can be applied for Close-to-Failure as well. This is an advantage as it allows for easy comparability of H-RAFT with C2F. Furthermore, combinations of H-RAFT and C2F may be considered (see section 7.3).

7.2. Variants

In section 7.1, the idea of the Close-to-Failure algorithm has been presented. Now, two variants of the approach are discussed. The first variant $C2F_{PART-F}$ requires only a minimum model knowledge. $C2F_{PRED}$ increases the flexibility of rule specification.

7.2.1. C2F$_{\mathsf{PART-F}}$

The basic idea of C2F$_{\mathsf{PART-F}}$ is partitioning the final rule f yielding the rules of $\mathcal{E}$. If a user is not able to specify properties, for example if he lacks in-depth knowledge of the model, this C2F variant provides an alternative. The user is no longer required to define the e_i "from scratch", but he may exploit the fact that a final rule f is already specified. The e_i can be derived from the final rule by splitting f into subrules. For example, let f represent the comparison of a consistency vector after protocol execution. This rule may be split such that any changes in the consistency vector are considered "critical". The rules e_i thus represent a single change each. It may even be possible to derive the e_i automatically. While f evaluates to a Boolean value, the functions of $\mathcal{E}$ result in a weight. Thus, apart from splitting f, a relevance for each e_i has to be specified. In the most simple case each e_i will be assigned the same relevance. In case of automatic splitting, this will be the default, however the user may be given the chance to adapt those weights if he considers different weights appropriate.

If this C2F variant is chosen, it is not necessary to evaluate f *and* the e_i as they express the same property. f may thus be specified as a function over the e_i.

An evaluation of C2F$_{\mathsf{PART-F}}$ is provided within chapter 13. This includes comparison with the basic C2F approach.

7.2.2. C2F$_{\mathsf{PRED}}$

The second variant allows consideration of the predecessor global state within the specification of the e_i. Thus changes of variable values from one state to the next one can be tracked and

specified as a property. Although such properties could be defined without direct access to the predecessor state already, this variant allows for easier specification and thus for better readability making rule definitions less error-prone. This simplification is also implemented and can be used for the experimental evaluation of the Close-to-Failure approach in chapter 13.

7.3. Combination with H-RAFT

So far, the weight calculation for the Close-to-Failure algorithm has been discussed. As already indicated, Close-to-Failure and H-RAFT may be combined as they are both based on weights and follow the two-step selection method. Combination of weights for the global state selection and for the transition weight calculation is discussed throughout this section.

Global State Selection. The weights of the global states as defined in H-RAFT can be combined with the one introduced for the C2F approach. Thus, user-defined criteria can be combined with static ones. Both methods should profit from the combination. Including (valuable) user knowledge might improve the performance of H-RAFT. The static weights of H-RAFT may improve the performance of C2F if little user knowledge is provided or if the knowledge is not "valuable". Thus, it may represent a correction of misleading user information. Different combinations are evaluated in section 13.

Transition Selection. Transition weights of H-RAFT and C2F may be combined as well. As C2F allows a more subtle definition of weights − per variable, signal, timer a different weight is possible − those weights could be preferred. Each action element that is not assigned a weight through C2F may be assigned a static weight as in H-RAFT. Figure 7.1 gives an illustrative example for a transition weight calculation in C2F, H-RAFT and their combination. Let the weights be defined as follows:

C2F:
$w(t1) = 20;$
$w(t2) = 2;$
$w(var1) = 17;$

H-RAFT:
$wInput(signalWithoutParameters) = 100;$
$wAction(timerSet) = 4;$
$wAction(varChange) = 1;$

The overall transition weight is calculated by extending the input weight by the action weights applying function $f3_{action}(tr_j)$ (see page 70). The combined weight is calculated by using the C2F weight if defined. Otherwise, the H-RAFT weight is selected.

The first column of figure 7.1 contains the transition elements. In the second column, the C2F weights for the respective elements are displayed. The H-RAFT weights are shown in column three. The last column contains the weight derived for the combination of the C2F and H-RAFT weights. The last row displays the overall transition weight resulting for the respective approach.

The performance of different combinations is discussed and evaluated in section 13.

Transition Element	C2F	H-RAFT	Combination
`input sig;`	0	100	100
`set(7,t1);`	20	4	20
`set(17,t2);`	2	4	2
`task var1 := 17;`	17	1	17
`task var2 := 5;`	0	1	1
Transition Weight	39	110	140

Figure 7.1.: Example Weight Calculations for C2F, H-RAFT and Combination.

7.4. Summary

Throughout chapter 7, the Close-to-Failure approach has been presented. It is highly based on user-defined criteria. Thus, it complements the H-RAFT algorithm. Rules and subrules may be specified to express criticality of a global state within the reachability graph. The criticality expresses the users notion on how close the system is to a fault-tolerance property violation. In order to maintain comparability with the H-RAFT algorithm, the two-step selection process for the transition to fire next is applied. Two variants have been presented: The first one reducing the need for model knowledge at the cost of less expressive rules and the second one allowing specification of changes from a global state to its successor. Finally, it has been discussed how C2F and H-RAFT may be combined. An experimental evaluation follows in section 13.

Part III.

Tool

8. RAFT

Inclusion of the novel algorithms for reachability analysis and the improvement techniques presented throughout part II into existing tools, for example SDT, is hardly feasible. In commercial tools, the source code is usually not available. Tools where alterations of the source code are possible are designed and efficiently implemented for their special purpose. Thus, inclusion of novel approaches is too difficult. Therefore, an additional contribution of this thesis is the *RAFT* tool. The generic nature of *RAFT* allows for adding new algorithms and extensions easily. The purpose of the tool is only to provide an environment for the novel algorithms and to allow for experiments − including comparisons to existing algorithms. Thus, elements like a graphical user interface or convenient input methods for weights and other parameters are considered only marginally. Section 8.1 gives an overview of the tool. The features of *RAFT* are described throughout section 8.2. A short summary on the usage of the tool is provided in section 8.3.

8.1. Introduction

RAFT uses the "pure" SDL model as input. No extensions to the SDL language are required. The tool consists of two main parts shown in figure 8.1: the *RAFT-Parser* and the *RAFT-Validator*. The *RAFT-Parser* is shortly presented in section 8.3.1. It converts models specified in the textual notation of SDL into Java code. The *RAFT-Validator* performs the reachability analysis on the created Java code. The user may specify *parameters* like selection of the validation algorithm, maximum runtime etc. In section 8.3.2 a short summary of the available parameters is given. Furthermore, the fault-tolerance properties to be checked (the *final* rule or the subrules of C2F) can be specified in *user-defined rules*. Details on the supported user-defined rules are provided in section 8.2.5. Both, the parameters and the user-defined rules, guide the reachability analysis in addition to the selected algorithm. *RAFT* provides the three algorithms of SDT − exhaustive, bit-state and random − as well as the novel *H-RAFT* (chapter 6) and *C2F* (chapter 7) algorithms. Details on the implementation of the algorithms are summarized in section 8.2.1. Currently the output is displayed in a concise way that allows for easy evaluation of results for the purpose of investigating the suitability of the novel algorithms. However, this representation is not user-friendly, thus an interface (indicated by the dashed rectangle in figure 8.1) to provide the output of *RAFT* as message sequence charts (MSCs) [ITU93a] is also available. If required, this module may be implemented. An *MSC-Viewer* as well as an evaluation plug-in (*MSC-Query*) are included. Furthermore, the current state of the validation could be stored by an O/R-Mapper (for example [Hib]). This allows for continuing any aborted runs at a later time. An interface for such a tool is also provided.

RAFT is implemented in Java for portability and modularity reasons. Thereby providing comfortable support for easily integrating new exploration algorithms, specifying different methods

of time progress and selecting dedicated processes, for example processes implementing faulty components etc.

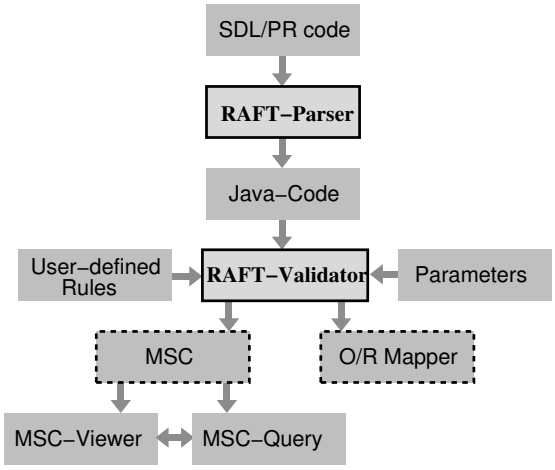


Figure 8.1.: Execution Sequence of *RAFT*.

8.2. Features of RAFT

The *RAFT* tool provides a variety of features for efficient reachability analysis of fault tolerance protocols. This section discusses the following features implemented in *RAFT* and their improvements with respect to the wide-spread SDT tool:

- The three standard algorithms: exhaustive, bit-state and random exploration as well as the two novel algorithms H-RAFT and C2F for reachability analysis (section 8.2.1);
- Partial order reduction through SFR-PO for shrinking the state space without loss of "interesting" parts (section 8.2.2);
- Extended definition of global states for capturing more behavior, especially in the time-domain and support of different time-progress models (section 8.2.3);
- Efficient handling of initializing transitions and support of fault-location and fault-evaluation processes outside the protocol model execution as well as definition of special processes representing faulty components (section 8.2.4);
- Specification of user-defined rules to express (fault tolerance) properties of the protocol and for restriction and further guidance of the exploration. Also definition of additional abort and restriction criteria for the validation runs (section 8.2.5).

8.2.1. Implemented Algorithms

SDT provides three algorithms for reachability analysis. For comparison, these algorithms have also been implemented in *RAFT* as well as the two novel algorithms. This section gives a short summary on the respective implementations:

Exhaustive Exploration. The exhaustive validation method as described in section 3.2.1 performs a depth-first traversal of the reachability graph. The algorithm itself is implemented in *RAFT* as it is in SDT: a simple depth-first algorithm. However, the definition of a global state in *RAFT* is more comprehensive than in SDT (see section 8.2.3).

Bitstate Exploration. The bitstate algorithm of *RAFT* is also defined based on the exhaustive algorithm as it is in SDT (see section 3.2.3). It differs from the SDT version as, by default, *RAFT* uses a single CRC-based hash code function, while SDT uses two different hash functions. Using one hash function only reduces computation time and increases the allowed maximum size of the hash table (which is limited by memory space) as only a single hash table is required. The larger portion of distinguishing elements of global states (see section 8.2.3) justifies the use of a single hash function. Furthermore, experiments have shown, that less than 0.1% of all detected reconvergences were unjustified. Yet, *RAFT* also provides a second (simple) hash function. Together with the CRC-based function, this simple function can be used to enable the *SDT* strategy of using 2 hash functions (and tables) for reconvergence detection.

Random Exploration. The random algorithm of SDT resembles a set of simulation runs. A single path to a global state where no progress can be made or to a reconvergence is generated. The path is created by randomly selecting the transition to be fired next. The algorithm can be repeated for a specified number of times. Yet, there is no guarantee that each path is generated only once. In *RAFT*, the algorithm is implemented alike.

H-RAFT. The H-RAFT algorithm is implemented as presented throughout chapter 6. Different depth, width and weight-calculation functions are provided. Furthermore, an interface is provided to include additional functions easily. Weights are specified through parameters.

Close-to-Failure. In chapter 7, the Close-to-Failure algorithm has been presented. It is also included in *RAFT*. Again, different functions for weight calculation are included and the weights may be specified as parameters. Furthermore, the user may specify his close-to-failure criteria. Those criteria will be applied by *RAFT* to guide the exploration.

Additional algorithms for reachability analysis can be implemented easily in *RAFT* as the tool uses a modular concept. It provides methods for arbitrary walks through the state space. The programmer is only required to specify the desired criterion for selecting the next transition to be fired within the global state space.

8.2.2. Partial Order Reduction

The single fault region partial ordering *SFR-PO* approach as introduced in section 5.1 is implemented in *RAFT*. In the parameter class (see section 8.3.2), detailed specification of single fault regions is possible.

8.2.3. Global State Definition and Timing

Global states in SDT are defined by the current local states of all processes, their variable values, active timers and input queues. In *RAFT*, it is not only differentiated between a timer being active or not, but also between different expiration times. In fault-tolerant systems it is often the case that a timer is defined only once, but set and reset repeatedly. For example a single timer representing a maximum delay between events. Thus, it is also important to differentiate between the absolute expiration times of a timer. Otherwise, behavior may be missed.

An additional element of distinguishing global states in *RAFT* is the current model time which is not considered in SDT. The SDT tool sets the variable *now* − representing the current model time − to zero for the complete analysis. Thus, comparisons with *now* may be evaluated wrongly during an exploration run. In *RAFT* this weakness is eliminated and those comparisons yield correct results.

In *RAFT* as well as in SDT, time is not handled continuously, but advances in (possibly non-equidistant) units if a timer expires. Model time then advances to the expiration time of that timer (for details see section 5.1.3). Although the state space size increases to some extent by including model time into the global state as is done by *RAFT*, omitting model time may be critical as it may lead to improper reconvergence detection, for example if the fault tolerance protocol under consideration executes the same tasks cyclically. If two cycles are executed identically, a reconvergence would be detected by SDT and the path will no longer be followed. A fault occurring in later cycles would not be discovered. By extending the global state definition of SDT through inclusion of current model time, faults occurring in later cycles are detected by *RAFT*.

The timing model "Time Progress by Timer Expiration" (section 5.1.3, page 48) has been implemented in *RAFT* for comparability with SDT. Yet, an interface for symbolic processing of time (see section 5.1.3, page 52) is included in *RAFT*. Furthermore, automatic generation of a ticker process as introduced in section 5.1.3 (page 51) can be easily implemented through an interface.

8.2.4. Special Transitions and Processes

Start Transitions. As motivated in section 5.2, start transitions in *RAFT* are executed in the order of process IDs. Thus only a single execution order is applied.

Specification of a Fault Location Process. The fault location process as described in section 5.3 selects the respective faulty processes for each scenario. Fault location processes are fully supported by the *RAFT* tool, thus reducing the number of concurrent events during reachability analysis. The process has to be specified in the parameter class (see section 8.3.2).

Specification of an Evaluation Process. In *RAFT*, a dedicated evaluation process (see section 5.3) is supported. Like the fault location process, it is not part of the protocol model, but is executed afterwards. Thus, the analysis sequence in *RAFT* is:

1. Initialization: Execution of start transitions.
2. Fault Location: Execution of the fault location process.
3. Analysis: Execution of the protocol model.
4. Evaluation: Execution of the evaluation process.

Specification of Faulty Processes. *RAFT* provides a parameter to specify the SDL processes that represent faulty components. Thus, spontaneous transitions in those processes may be considered explicitly, for example. This approach has been introduced in section 6.3.1 (page 65).

8.2.5. Rule and Criteria Definition

User-defined Rules. Throughout section 7.1, a distinction between the final rule f and subrules $e_i \in \mathcal{E}$ has been discussed. The final rule denotes the fault-tolerance properties of the protocol that are subject to investigation. Through the e_i rules used for weight calculation by the Close-to-Failure algorithm are denoted. Both kinds of rules can be subsumed by the term "User-defined Rules". Those user-defined rules may be specified to check whether certain properties of the protocol are fulfilled or violated. In *RAFT*, any rule can be defined since access to all global states, including timers and current system time is possible. Properties may be based on the current global state, or even on previously visited global states. Typical properties to be checked comprise:

- Is process A in state s_i?
- If process A was in state s_x at time t_1, is it in state s_y before time t_2?
- Is variable v of process s_i ever assigned value a?
- Did at least 2 out of 3 processes receive the same signal within a time-interval Δ?

These properties are specified through expressions over global states.

Apart from the rule definitions as provided by SDT, *RAFT* allows also for specification of multiple rules and inclusion of system time in the questions. Furthermore, event sequences can be checked. This results in a highly flexible and very powerful rule definition scheme and is fully supported by *RAFT*.

Abort Criteria. *RAFT* provides several criteria for stopping or restricting an exploration:

- **Time Limit:** The user may specify a run-time limit. The exploration is aborted if the limit has been passed after execution of a transition. Time limits are defined with a resolution of milliseconds. This criterion can be used with any algorithm. It is not available in SDT.
- **Depth Limit:** If a depth limit is specified by the user, exploration of a path is stopped once the limit has been reached. This criterion limits the exploration of the state space to some depth. Note that, different from the time limit, it is not the exploration run that is aborted if the criterion is fulfilled, but the current path is left and exploration continues with a different path. Depth limits can be defined for all algorithms in SDT and in *RAFT*.
- **CTRL-C:** The exploration run can be stopped manually by killing the respective process. In contrast to SDT, the results found so far will remain available.
- **User-defined Rule:** The user-defined rules as described in the previous paragraph can be applied in two ways: On fulfillment of the rule, either the current path is cut, but the global exploration continues, or the complete exploration is terminated. In both cases, the paths resulting in the fulfillment of the rule can be stored for off-line analysis. It is also possible to specify (different) rules for aborting the analysis and those for leaving the path in a single exploration run. SDT supports only a single rule. For this rule it has to be specified whether an abort of the path or of the exploration should occur.

8.3. Usage of RAFT

In this section a short overview on the *RAFT*-Parser and the parameter class as shown in figure 8.1 is given.

8.3.1. RAFT-Parser

The *RAFT*-Parser converts SDL/PR models into Java classes. It supports all major SDL constructs. The currently supported language elements are summarized in table 8.1. Not supporting elements like macros does not limit the use of the language as they are for convenience only. Other elements − like enabling conditions − are usually rarely used in SDL, but can be added easily if needed. The available language elements already cover a large subset of SDL. This subset has been sufficient for modeling the very large industrial FlexRay model (see section 10.7).

Each SDL process is converted into a separate Java class specifying the program flow of the automaton expressed by the respective process. The generated classes are human-readable and may be edited (for model debugging purposes etc). As an example, figure 8.2 shows an SDL process `proc_A` and the respective generated Java code in figure 8.3.

User-defined SDL data types are also converted to Java classes. These classes provide methods as required for the respective types and the defined variables.

Supported	**Not supported (yet)**
System, Block, Process	System-/Block-/Process-Type (OO-Features)
Channel, Signalroute	Block/Channel Partitioning
Signals (also with parameters)	Continuous Signal
Variables	Viewed/Revealed; Imported/Exported Variable
Actions: Task, Output, Decision, Join, Stop	Actions: Create, Call, Return
Nondeterministic Decision	Enabling Condition
Datatypes: Syntypes, Synonyms, Structure Sorts	Datatypes: External Synonyms
Generators : Array, String	Generators: Powerset
Input, Save, Spontaneous Transition	Priority Input
Asterisk Save, Asterisk Input, Asterisk State	(Remote) Procedures
Connection (of Channel and Signalroute)	Service and Package
Timers, Timerarrays	Macros

Table 8.1.: Supported SDL Language Elements.

```
process proc_A;
      Timer t_low_A, t_high_A;
      start;
            set(5,t_low_A);
            set(25,t_high_A);
            nextstate wait_low;
      state wait_low;
            input t_low_A;
                  nextstate send_sig;
      endstate wait_low;
      state send_sig;
            input none;
                  reset(t_high_A);
                  output sig;
                  stop;
            input t_high_A;
                  output sig;
                  stop;
      endstate send_sig;
endprocess proc_A;
```

Figure 8.2.: *RAFT*-Parser: Example SDL Model.

```
public class T_Process_0 extends T_Process {   /* proc_a */
   private boolean t_low_a = false;
   private boolean t_high_a = false;

   public void executeTransition(T_Signal signal) {
      if(signal.getID() >= T_Constants.timerSignalOffset+T_Constants.numTimers){
         return; }
      switch (localStateID){
         case 0: { /* start */
             setTimerAbsolute(TP_Typecheck.string2Double(""+5) ,
                 (short)(1 - T_Constants.timerSignalOffset)); /* t_low_a */
             t_low_a = true;
             setTimerAbsolute(TP_Typecheck.string2Double(""+25) ,
                 (short)(2 - T_Constants.timerSignalOffset)); /* t_high_a */
             t_high_a = true;
             localStateID = 1;   break; } /* endstate start */
         case 1: {    /* wait_low */
            switch(signal.getID()){
               case 1: {   /* begintrans t_low_a */
                  t_low_a = false;
                  localStateID = 2;   break;
               } /* endtrans */
            }
            break; } /* endstate wait_low */
         case 2: {    /* send_sig */
            switch(signal.getID()){
               case -1: {   /* begintrans none */
                  resetTimer((short)
                  (2 - T_Constants.timerSignalOffset)); /* t_high_a */
                  Object[] p1 = null;
                  sendSignal((short)1, (short)0,p1);
                  localStateID = - 1;   break; } /* endtrans */
               case 2: {   /* begintrans t_high_a */
                  t_high_a = false;
                  Object[] p3 = null;
                  sendSignal((short)1, (short)0,p3);
                  localStateID = - 1;   break; } /* endtrans */
            }
            break; } /* endstate send_sig */
         case -1:{    /* stop */
            break; }
      }  // END Switch
   }  // END executeTransition
}  // END T_Process_0
```

Figure 8.3.: *RAFT*-Parser: Resulting Java Code.

Synonyms (representing constants) are translated into static final variables.

Parsing the SDL code is only necessary if changes have been made to the model. Otherwise, Java code generated once is stored and can be re-used for each exploration run.

8.3.2. RAFT Parameter Class

Parameterization of the validation runs is done in a parameter class. In *RAFT*, this class is automatically generated by the parser and contains default values (see figure 8.4). These values may require adaption for the specified model. They will be checked for consistency before the validation run starts.

```
public class T_Parameter {

     /* Constants for validation */
     public final static byte validationMethod = 3;
     // 0 - exhaustive
     // 1 - random
     // 2 - bitstate
     // 3 - h-raft
     // 4 - c2f
     public final static int maxDuration = 0;      // in ms
     public final static int maxDepth = 0;
     public final static int numRepetitions = 1;
     public final static int bitstateHashSize = 0; // in Bit

     /* Constants for partial ordering */
     public final static short[] decisionProcess = {3};
     public final static short[] evaluationProcess = {};

     public final static short[][] poGroups ={
      {0,1,2,4}
     };

     public static final short faultyProcess = 2;
}
```

Figure 8.4.: Parameter Class of RAFT.

In this file, the desired exploration algorithm, the maximum duration of the run and the maximum allowed depth of the state space are denoted. These constants can be set for all validation methods. Furthermore, the number of validation runs that should be performed is specified. This parameter may also be applied to each of the algorithms. It is most useful to specify the number of repetitions for the random algorithm. The size of the hash table for the bitstate algorithm (see section 8.2.1) is also defined here. If an algorithm other than bitstate is selected,

this parameter is ignored. With respect to the partial ordering strategy, the fault location, the fault evaluation and the faulty process(es) can be specified.

RAFT automatically generates a single partial ordering group containing all processes. This corresponds to not applying the partial ordering techniques. The user can split the processes into groups as required. If more than one partial ordering group exists, SFR-PO is applied in conjunction with the selected algorithm.

Part IV.

Analysis

9. Introduction and Goal of the Analysis

This part contains the experimental evaluation of the algorithms and methods presented in the previous part. The analysis can be split in two parts. The first goal is to determine suitable parameter and weight settings for the novel methods and algorithms that are applicable to all fault-tolerance protocols. Apart from parameters and weights, also different functions and combinations are investigated and compared to each other. Thus, the novel algorithms are investigated. The second goal − which is the main purpose of the experiments − is a comparison to existing algorithms. Thereby evaluating whether the novel approaches yield a performance increase. More detailed goals are provided in the respective sections.

In order to prove the generality of the results, seven protocols − ranging from rather small ones to a very large one − have been modeled for the experiments. For all models subtle design faults were inserted artificially (see chapter 10), leading to fault-tolerance violations if one of the participating components exhibits certain faulty behavior. After introducing those design faults, faulty behavior of components has been specified by using the "any output at any time" paradigm. This most universal fault-model comprises any faults in the value and time domain. For some of the protocols this paradigm has been restricted to achieve more deterministic behavior, thereby reducing the resulting state space.

For all experiments a run-time limit of 48 CPU-hours has been assumed. This limit represents the typical willingness of a user to wait for results and is a realistic assumption within industrial projects. If within that time no fault-tolerance violation has been observed, the experimental run has been aborted. Furthermore a memory limit of 1GB has been assumed as this is the typical equipment of todays PCs. If neither a violation of the time-boundary (which was the most common violation) nor of the memory limit has occurred, experimental runs have been stopped once a fault-tolerance violation has been discovered.

Organization of Part IV. Throughout this first chapter of part IV, an introduction to the experiments and their goals is provided. Chapter 10 gives a description of the modeled protocols including the inserted faults. Chapter 11 provides an analysis of the single-fault-region partial ordering approach. The results are verified by applying SFR-PO to all of the modeled protocols. The H-RAFT algorithm is evaluated in detail in chapter 12. Chapter 13 provides the investigation of the Close-To-Failure algorithm. A comparison of the results of the novel algorithms and those achieved by general algorithms (exhaustive, bitstate, random) is discussed in chapter 14 including a summary and conclusion of the experimental findings.

10. Modeled Protocols

For the experimental evaluation of the novel algorithms, seven fault-tolerant communication protocols have been implemented. The Pendulum Protocol (PP) [Ech87, Ech89] (see section 10.1), the Signed Messages Protocol (SM) [LSP80, Ech90] (section 10.2), the Randomized Byzantine Agreement Protocol (RBA1) [Tou84] (section 10.3), a deterministic variant of the RBA1 protocol (DBA1) (section 10.4), the VETO Protocol (VETO)[EM00] (section 10.5), the 2-Switch Protocol (2SP) [course "Modeling Fault-tolerant Systems" at the university Duisburg-Essen] (section 10.6) and the FlexRay Protocol (FX) [Fle02, BBB$^+$00] (section 10.7).

These protocols result in different sized models with respect to both the lines of code and the assumed global state space size. The Pendulum Protocol, the SM protocol and, especially, the DBA1 protocol result in rather small models. The models of RBA1, the 2SP and the VETO protocol are medium sized. FlexRay is a large-scale industrial protocol resulting in a huge model.

Small models allow for fast evaluation of promising parameter combinations. By investigating the performance of *H-RAFT* and *C2F* for highly complex fault-tolerant protocols, the benefits of the algorithms for application in practice are evaluated.

In the following sections, the protocols are shortly characterized and their inserted design faults as well as the behavior of their respective faulty component(s) are described. Each protocol is introduced by giving an overview over the underlying algorithm. A pseudo SDL model of the respective implementation is also presented. Furthermore, details on the inserted design fault and the experimental setup are provided. The general guideline for the inserted design faults is that they are hard to detect. Subtle changes are made to the original protocol such that faults occur under rare conditions only.

10.1. Pendulum Protocol

The Pendulum Protocol (PP) is an agreement protocol. Instead of a distribution protocol "message-ping-pong" is employed. The goal of the protocol is to use a minimum number of messages in the fault-free case and as little messages as possible in the faulty case. Although the number of messages is minimal, the protocol is rather slow as the number of (short) phases is always suboptimal. To allow for only $m \geq 2\dot{F} + 1$ nodes, signatures are required. After execution, the fault-free nodes deliver a value according to a distance-decision. In other words, the results of all fault-free nodes are in the δ-environment of the fault-free start-values. Each node executes a different part of the protocol, thus each node has to be aware of its index. Basically, F+1 pendulum strokes are required where the number of nodes per stroke decreases. Up to F consecutive faulty nodes must be by-passed. In order to detect unjustified (faulty) by-passes, the "by-passers" have to be by-passed again. Within each pendulum stroke, F distance-decisions have to be made. Thus, an F+1-times signed message is created. This message can be transferred to all other nodes simultaneously, thus only F+1 phases are required.

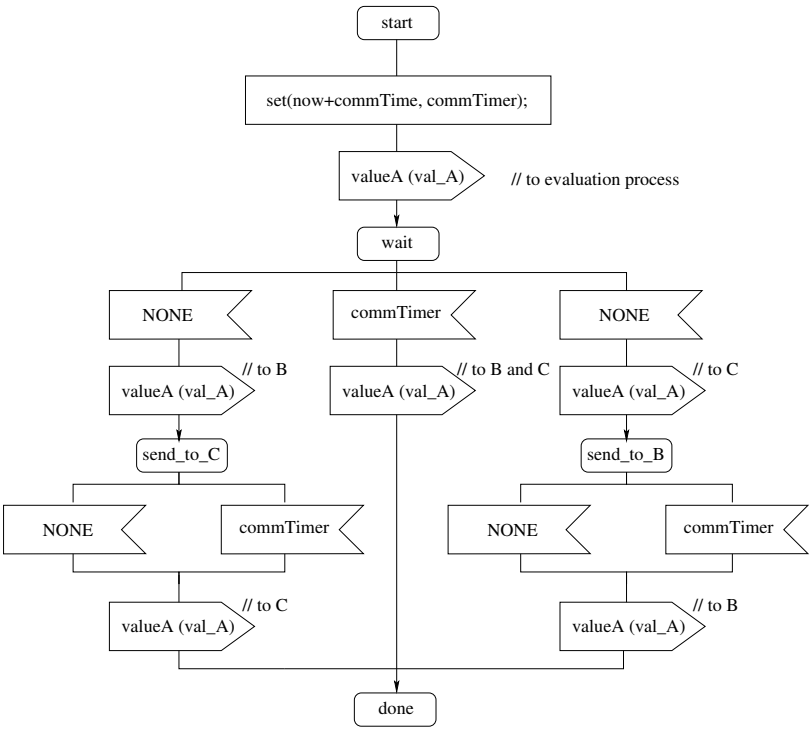


Figure 10.1.: Pseudo SDL Model of Node A of the Pendulum Protocol.

Figures 10.1, 10.2, and 10.3 show the pseudo SDL code of the Pendulum Protocol for three nodes as the three participating nodes execute different code.

Node A (figure 10.1) sends its value to nodes B and C during time interval [0, `commTime`].

Node B (figure 10.2) waits for `valueA` for a time interval. If `valueC` is received in this interval it is ignored. If either the interval timer expires or `valueA` is received and differs too much from `valueB`, node B waits for a value from node C. Otherwise, node B forwards the value received from node A to node C. If node B has to wait for `valueC`, it decides on `valueC` once received. If a value from node A arrives while `valueC` is expected, the value is ignored.

Node C (figure 10.3) waits for a value from node A or node B. If `valueB` arrives first, node C delivers this value (`valueB`) and finishes its protocol loop. If a value from node A arrives first, node C will wait for `valueB` during a time interval. If `valueB` arrives within this interval, this value is delivered. Otherwise `valueA` and `valueC` are compared. If the result is smaller than or equal to ε, `valueA` is delivered. Otherwise, `valueC` is delivered and sent to node B at an arbitrary point in time within a predefined limit. Finally, node C terminates.

Design Fault. The design fault of the Pendulum Protocol is inserted into node B. The duration of the first `commTimer` − set during the start transition − is selected slightly too small. More precisely, it is set to the minimum communication delay between node A and node B. Thus it is possible for `valueA` to arrive on time if only the minimum transfer duration is required.

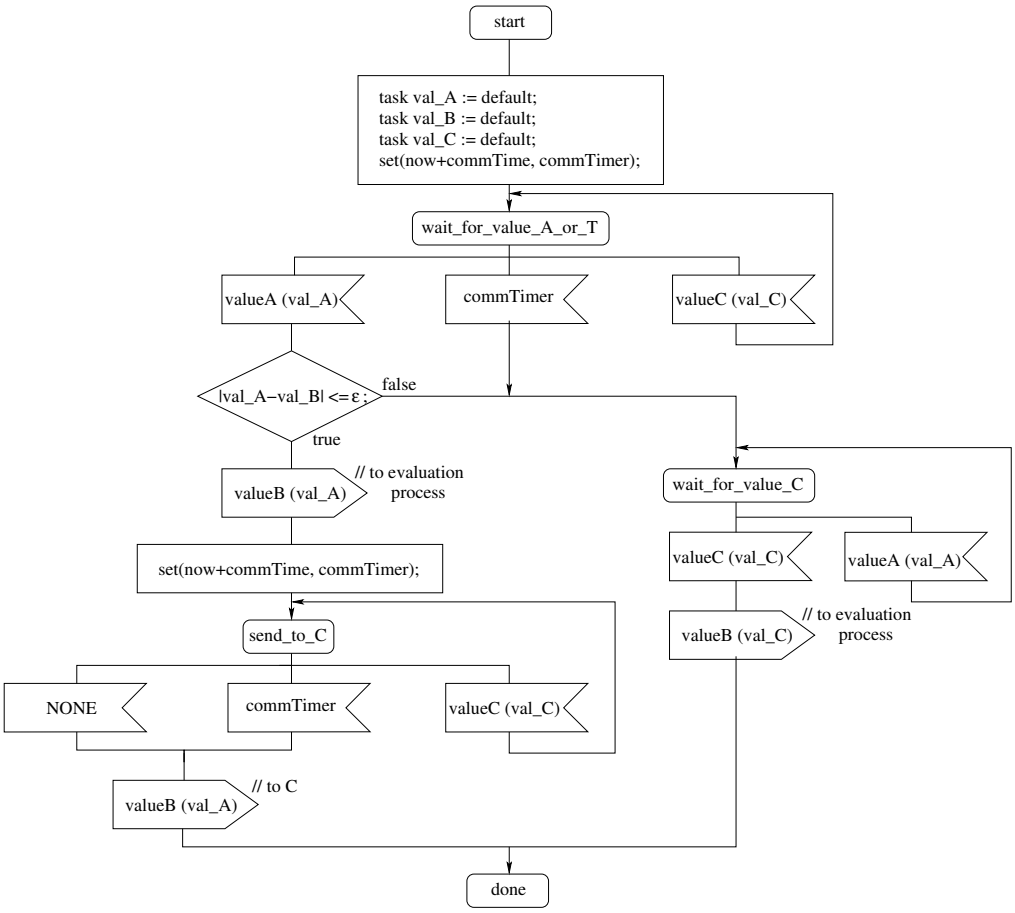


Figure 10.2.: Pseudo SDL Model of Node B of the Pendulum Protocol.

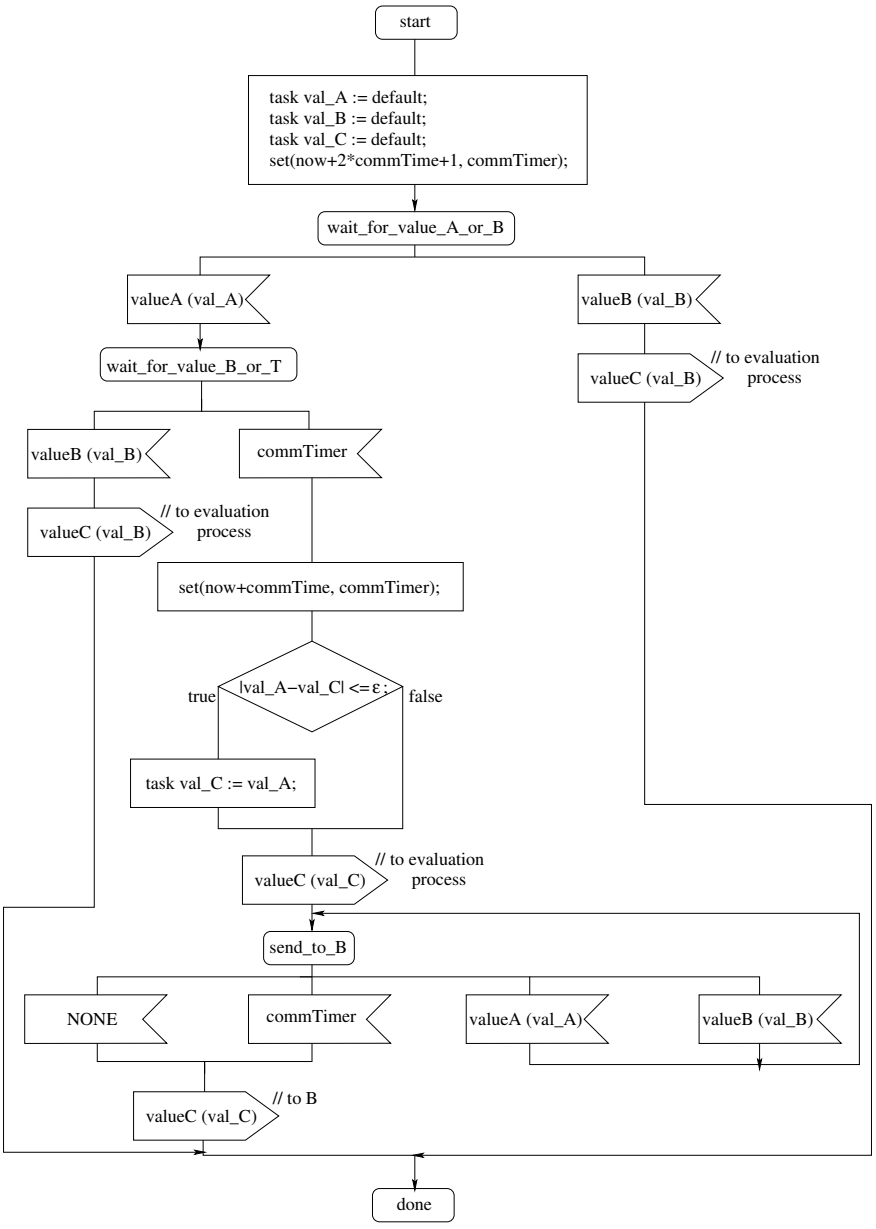


Figure 10.3.: Pseudo SDL Model of Node C of the Pendulum Protocol.

However, it is also possible that `valueA` is transmitted within the maximum communication delay (and would thus be valid), but is considered too late by node B.

Experimental Setup. For the experiments a single-fault assumption is made. Thus, three nodes are implemented. The experiments cover each node being faulty (in turn!). Faulty nodes may exhibit any faulty behavior in the time and data domain.

10.2. Signed Messages

The signed messages (SM) protocol is a distribution protocol. In SM_k k nodes exchange their values with each other resulting in a consistency vector of length k at every node. The consistency vectors of all fault-free nodes are equal after protocol execution. Protocol execution takes exactly $F+1$ phases, where F denotes the maximum number of faulty nodes. In the fault-free case, messages are only sent in the first two phases. Each node has to sign the value before sending. All forwarded messages are cosigned. Signatures cannot be tampered by faulty nodes without recognition of the fault-free nodes. Thus, Byzantine faults (= deviating values) can be discovered. If a deviation is detected, the source sending the initial value must have been faulty. This information has to be distributed to the other (fault-free) nodes. Then the faulty sender can be ignored and all fault-free nodes agree on a default value.

The pseudo code for the signed messages protocol is given in figure 10.4. First, all values of the consistency vector `cVector` are set to `unknown` and the own initial value `val` is inserted. Then the first phase is started by signing `val` and sending it to all other nodes. Now, the protocol enters the loop for another *F+1* phases. *F* denotes the maximum number of faulty nodes. Phases are limited by their duration `phaseDuration`. If the timer `phaseTimer`, indicating the end of the phase, expires in state `wait`, the phase counter `phase` is increased and the timer is set again. If a message from another node `senderId`, indicated by the input `message`, arrives during a phase, it is checked (`chkMessage(message)`) whether the value and the signatures are valid. For better readability, the exact checks are not shown in the figure. It is checked, whether the following expressions all evaluate to true:

- Are all signatures valid?
- Are all signatures from different nodes?
- Is the number of signatures at least the number of the current phase?
- Is the received value different from the one in `cVector(senderId)`? (This also means different from `undefVal`).
- Is the value of `cVector(senderId)` different from `multipleVal`?

If any of these checks evaluates to false, the message is ignored. Otherwise, `cVector(senderId)` is set according to the next decision: If the value of `cVector(senderId)` is still undefined, it is set to the received value `val`. If it is already set, it must have been set to a value different from `val` as equality has been checked in `chkMessage(message)`. Thus, `cVector(senderId)` is set

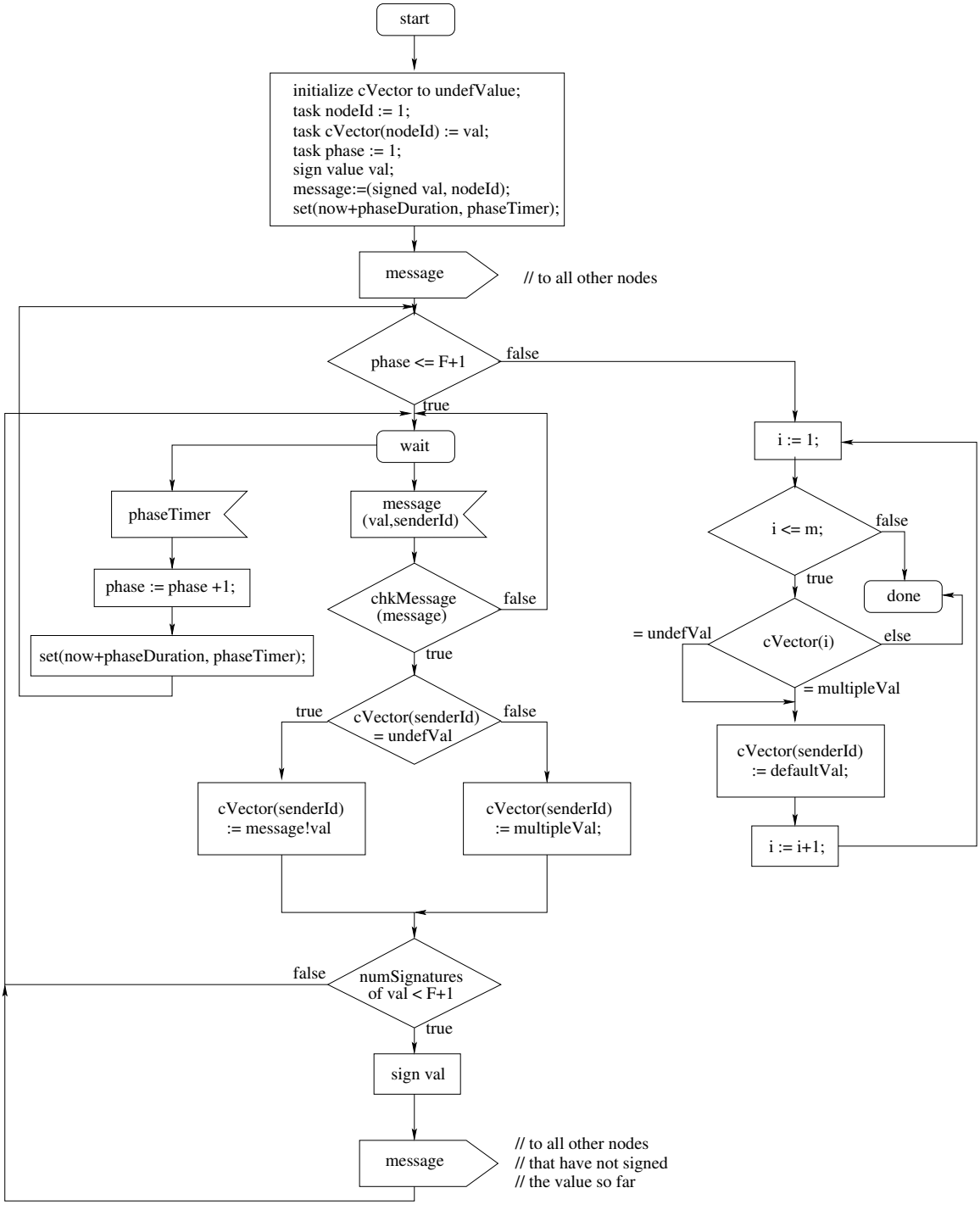


Figure 10.4.: Pseudo SDL Model of the Signed Messages Protocol.

to `multipleValues`. In both cases the protocol continues with checking whether the number of signatures of the received `val` is less than $F+1$. If this is the case, the signature of the node is added and the message is forwarded to all nodes that have not signed `val` so far. Then state `wait` is resumed and the next signal or the end of the phase is awaited. If there are at least $F+1$ signatures to `val`, the value is not distributed to the other nodes as at least one fault-free node must have signed it.

Once the loop over the phases is left, another loop over the elements of the consistency vector is entered. If the value of an element is still set to `undefVal`, or if it is set to `multipleVal`, then it is assigned the default value `defaultVal`. Otherwise, it is unaltered. The end of the loop is also the end of one protocol cycle.

After execution of the protocol, the decision for a single value can be accomplished by a median decision. At 2m+1 nodes, the m smallest and m highest values are discarded and the remaining value, that is the value "in the middle" is chosen. The median decision is not part of the protocol itself.

Design Fault. The design fault introduced into the protocol model is to omit one check in `chkMessage(message)`. It will not be checked, whether all signatures of a value `val` are from different nodes.

Setup. The model of the signed messages protocol implemented for the experiments consists of four nodes. One of them is faulty. The faulty node exhibits Byzantine behavior. It sends the same value as the fault-free nodes to 2 of the fault-free nodes. It sends arbitrarily the correct or a wrong value to the third fault-free node. Especially, it may also forward a value different from the one it received and can sign it multiple times.

10.3. Randomized Byzantine Agreement (RBA1)

The *R*andomized *B*yzantine *A*greement protocol is a probabilistic agreement protocol based on randomization. The variant considered here is the RBA*1* protocol, that is, a decision on a single bit is sought. The protocol is based on epochs consisting of two phases each. In each phase, the current values of the nodes are exchanged and a decision on a new current value is taken. If a node has a sufficient majority for one value it can conclude that the other nodes have a sufficient majority as well and may terminate. The basic algorithm in each phase is as follows:

During the exchange part, all nodes distribute their current value to all other nodes. In all even phases, a single node determines and distributes an additional random bit. In every even phase, another node to generate the random bit is cyclically selected. After each exchange, every node executes a most-frequent-value decision over the received values. In this decision, only "0" and "1" are considered. If an "unknown" value is received, this will be ignored. Furthermore, it is determined how often the most frequent value (MFV) has been received. This number will be referred to as *num* further on.

In each odd phase, the current value is set to *MFV* only, if *num* is a large majority. Otherwise, the current value is still "unknown".

In each even phase, the current value is set to *MFV* even if it is only a small majority. If the small majority is not achieved, the current value is set to the random bit value. In case of a large majority, the final value is also set to the current value and the algorithm terminates two phases later. These additional phases ensure that all other fault-free nodes terminate as well.

The pseudo code for the RBA1 protocol is given in figure 10.5. On the left hand side − starting with expiration of the `roundTimer` − calculations and comparisons that have to be executed after each round are depicted. The code for execution after an even round and an odd round differs (`decision evenRound`) as described in the previous paragraphs. The right hand side provides the code for message reception and termination of the protocol.

Design Fault. The design fault introduced in this model is not to check the origin of the messages. Thus more than the expected values may arrive at a node tampering the decision on the selected value.

Setup. The protocol is implemented for four nodes. One of the four nodes is faulty. It randomly chooses "0" or "1" to be sent to the other nodes in each round at arbitrary times, thereby emulating faults in the value and in the time domain.

10.4. Deterministic Byzantine Agreement (DBA1)

The protocol *D*eterministic *B*yzantine *A*greement is a sub-variant of the RBA1 protocol (see section 10.3). Instead of distributing a random number, each node will use a predefined static value. Thus, each epoch consists of only a single round. In the absence of faults, the protocol terminates in the same way as RBA1. However, in the presence of faults, the protocol may not terminate at all.

The pseudo code for the DBA1 protocol is given in figure 10.6. Again, the left hand side shows the actions after each round including termination of the protocol. In contrast to RBA1, no distinction between odd and even phases is required. The right hand side depicts signal reception and distribution of the values.

Design Fault. The design fault is already included in the protocol: A default "random" value is predefined instead of generating a random number.

Setup. The DBA1 protocol is implemented for four nodes. One of the four nodes is faulty. While all three fault-free nodes always select "0" as random value, the faulty node randomly chooses "0" or "1". Only faults in the value domain are considered. In other words: the faulty node may send an arbitrary value, but at the correct time.

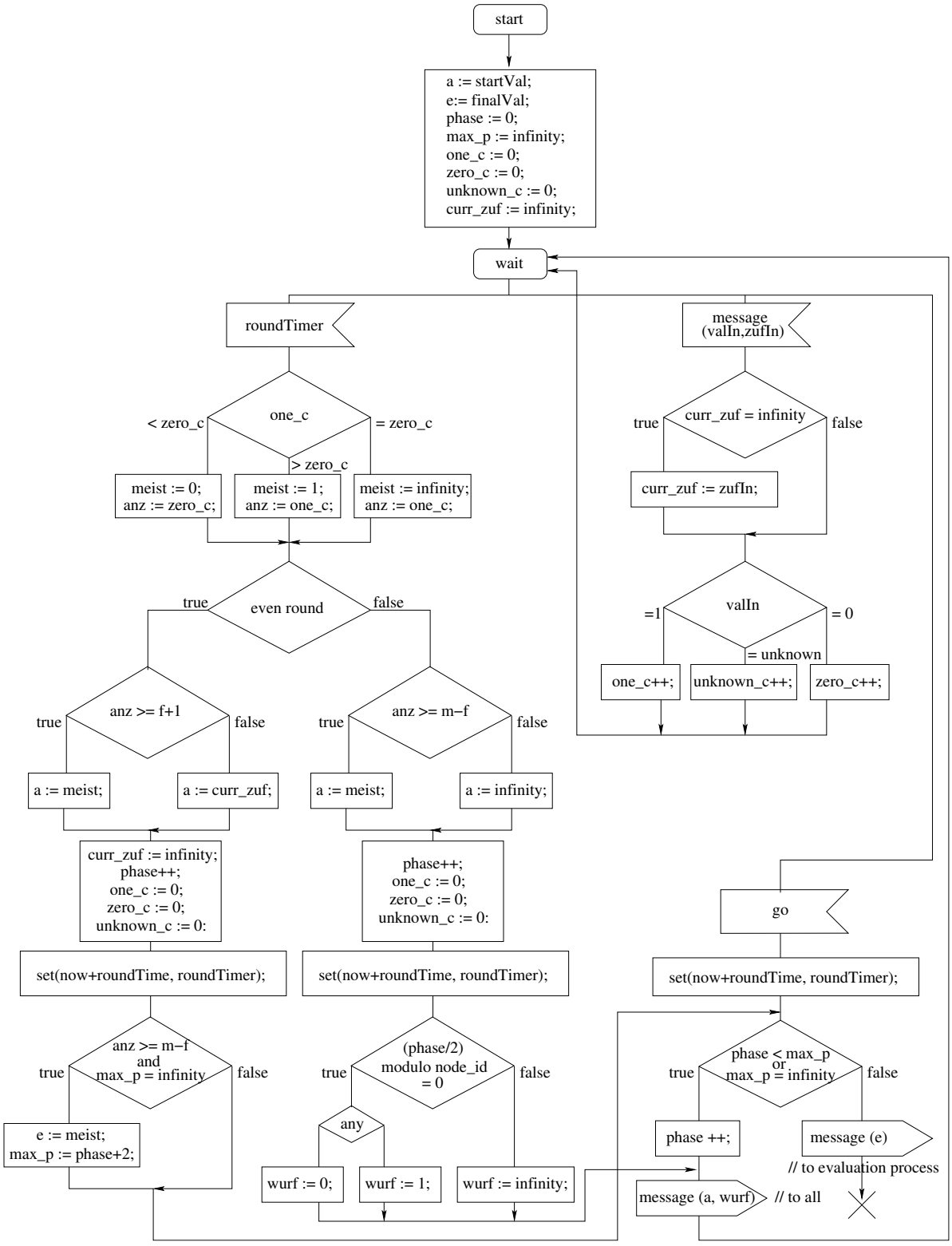


Figure 10.5.: Pseudo SDL Model of the RBA1 Protocol.

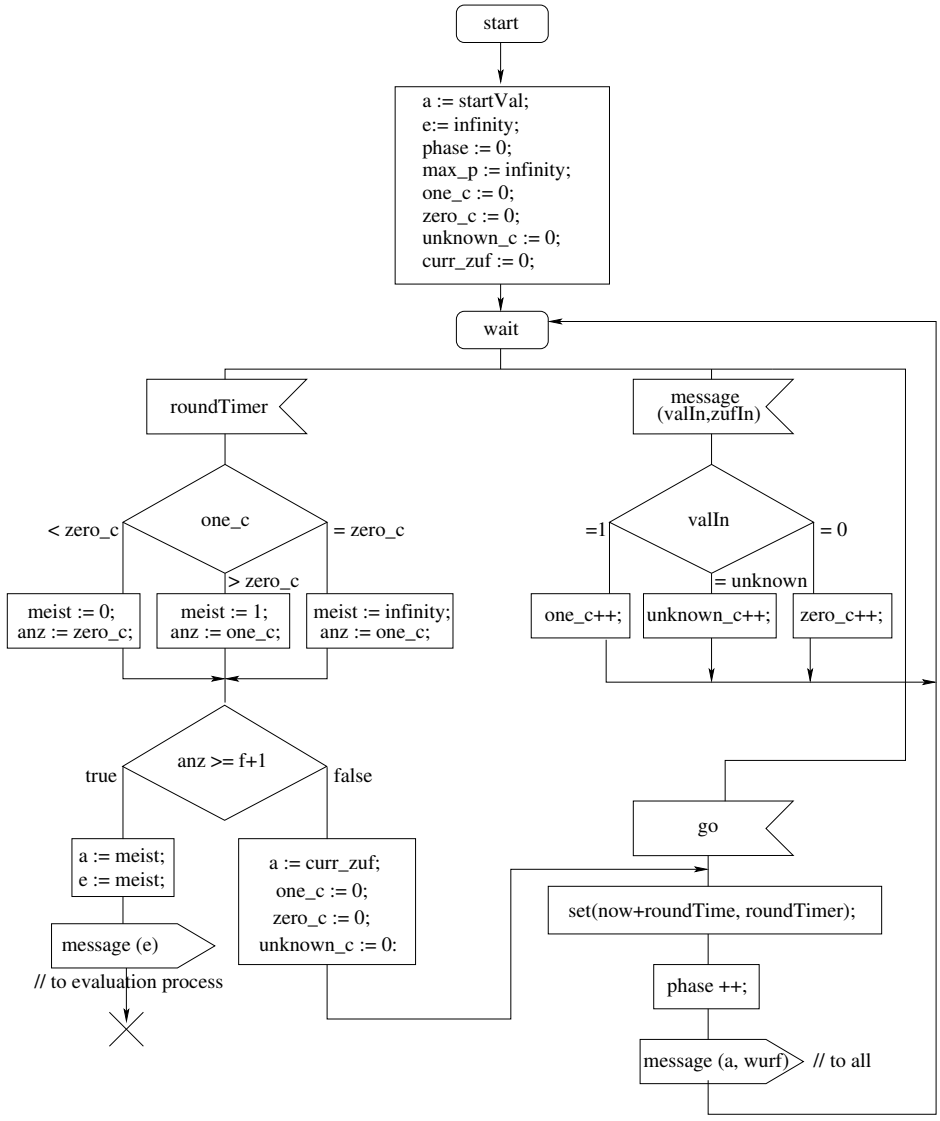


Figure 10.6.: Pseudo SDL Model of the DBA1 Protocol.

10.5. VETO Protocol

The VETO protocol is a multiple bus broadcast protocol resilient to non-cooperative Byzantine faults. In the faultless case performance is optimized while a wide fault model is supported. Except for cooperative Byzantine faults, all fault types are considered. Communication is based on identical message transfer on all busses in parallel. In case of message loss, some receivers regenerate the message. If values on different busses deviate, this is detected and they are spread system-wide. Thus, inconsistent delivery can be prevented. This induces overhead, while in the faultless case no timeout delays the execution. Hence, deviations may occur even after the message has been delivered. If a receiver that has already delivered receives another message, it enforces its value by sending veto messages to the other nodes. If several receivers issue a veto, a decision strategy has to be applied to prevent inconsistencies.

The veto technique is similar to a negative acknowledgment instead of a positive one. In the absence of faults it saves both time and messages.

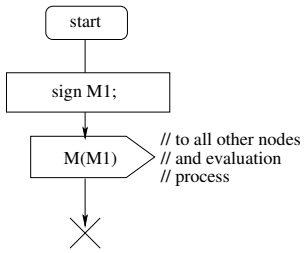


Figure 10.7.: Pseudo SDL Model of the Sender of the VETO Protocol.

The pseudo SDL code for the VETO protocol is given in figures 10.7 and 10.8. Figure 10.7 depicts a broadcast sender which simply signs and then broadcasts the message to all other nodes.

The protocol for the receiver is shown in figure 10.8. In the fault-free case, the receiver waits in state `listening` until it has received messages via all busses. Then it delivers the value and waits for some time (`tDelete`) whether or not more messages arrive. If no extra-messages are received, the process terminates. In case of a detected deviation during protocol execution, a veto is issued and the value is not delivered unless a double-signed veto is received.

Design Fault. The design fault included in the VETO protocol is to skip the check whether a newly received value is equal to the already received one(s) in state `delivered`. If an additional message arrives in this state, it is only checked whether a single signature is present or an additional signature has been added. A faulty sender may exhibit arbitrary Byzantine behavior. Thus, it may send correct values to one subset and wrong values to the complementary set. The result may be an unjustified veto.

Setup. The experimental setup of the VETO protocol is shown in figure 10.9. The four nodes N1, N2, N3, N4 are connected by three buses B1, B2, B3.

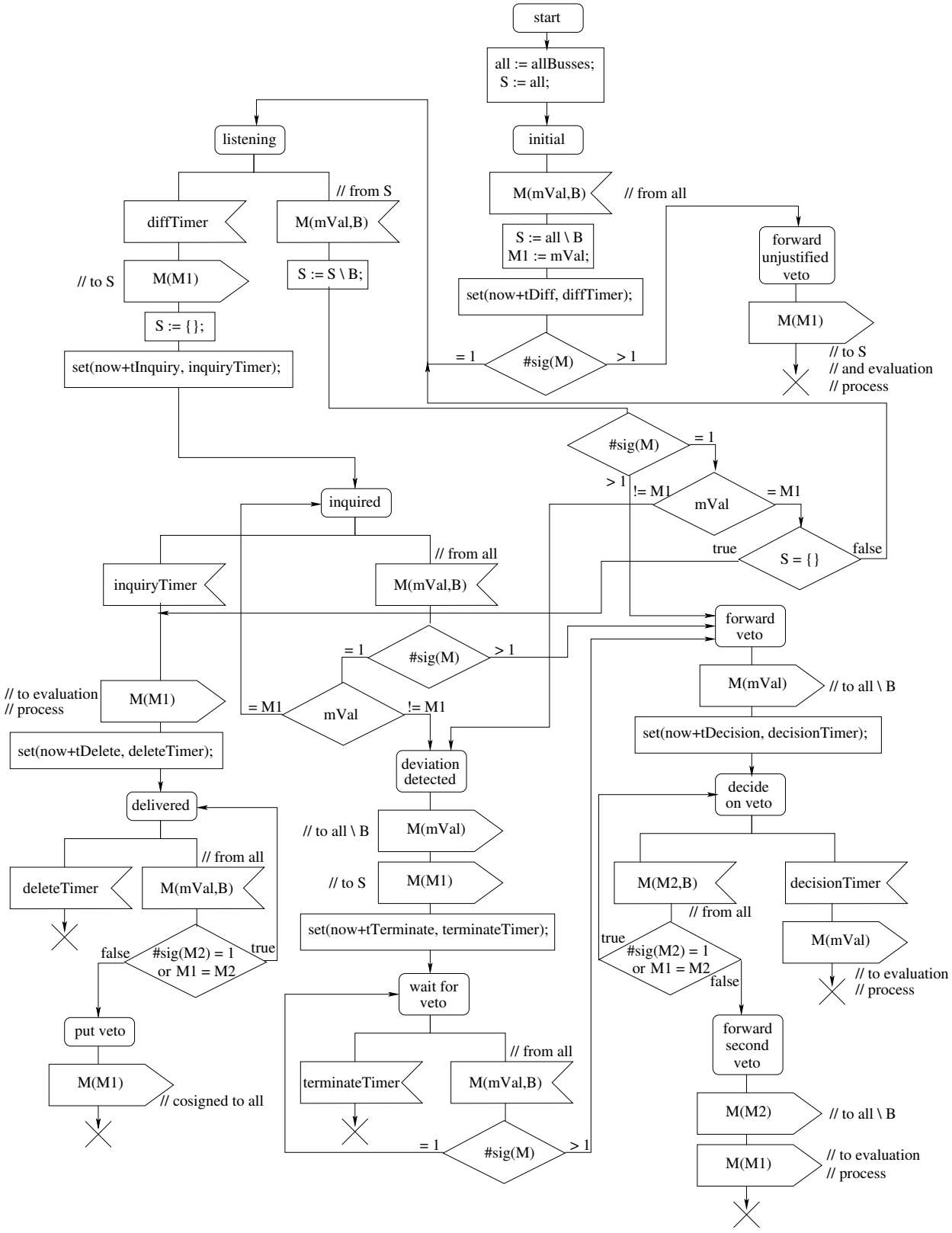


Figure 10.8.: Pseudo SDL Model of the Receivers of the VETO Protocol.

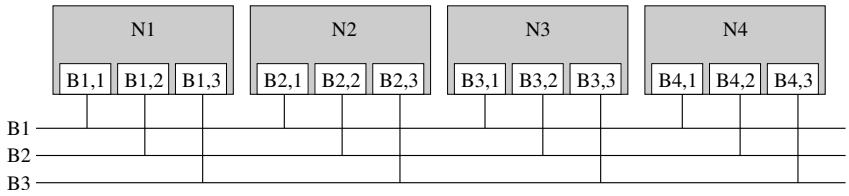


Figure 10.9.: Setup of the VETO Protocol.

10.6. 2-Switch Protocol

The 2-Switch Protocol (2SP) is a special protocol ensuring fault-tolerant message transfer from an input channel to an output channel. The system consists of 3 nodes (R1, R2, R3) and 2 switches (S1, S2) connecting an input channel (EK) and an output channel (AK) as shown in figure 10.10.

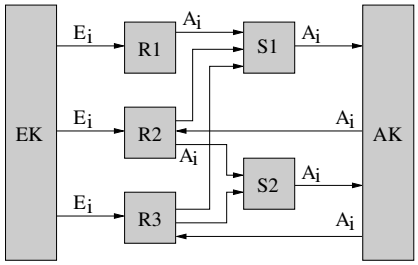


Figure 10.10.: 2SP Component Overview.

The input channel periodically transmits the values E_1, E_2, E_3, ... of the input signal (see figure 10.11).

A node R_k (figures 10.12, 10.13, 10.14) may receive input from the input channel and the output channel. It may send to the output channel, calculate the output value A_i to the corresponding input value E_i, compare values, decide on "correct" or "wrong". Furthermore, it may drive the switches by sending a binary signal "on" or "off".

A switch S_j (figures 10.15, 10.16) consists of a data input, a data output and control input (= CI) channels. At startup, all control inputs are set to "off". If all CIs are "off", no information is provided at the data output. Any incoming signals are lost. If at least one CI is set to "on", a value at the data input is forwarded via the data output.

The specific behavior of the respective components is defined as follows:

R1: receives values E_i from EK and sends output values A_i to S1. The pseudo SDL code of R1 is provided in figure 10.12.

R2: receives output values A_i from AK. Furthermore, it accepts values E_i from EK iff it determines that R1 has not sent its output to AK on time. This can be concluded from reception of the respective A_i from AK. By comparison of its own output value with the

value received from AK, it determines whether a value is considered "correct". If R2 decides that R1 has sent a "wrong" value, it sends "off" to S1, otherwise S1 is driven to "on". Figure 10.13 depicts the pseudo SDL model of R2.

R3: receives output values A_i from AK. Furthermore, it accepts values E_i from EK and calculates output values for internal comparisons. However, these values are never sent. If R3 determines − through comparison with A_i − that R1 has sent a "wrong" value to AK, it drives S1 to "off" and S2 to "on". Otherwise, it sends "on" to S1 and "off" to S2. Decision on the origin of the A_i (either R1 or R2) is based on the arrival time of A_i. The pseudo SDL code of R3 is shown in figure 10.14.

S1: receives values from R1 via its data input channel. It forwards this value to AK. Additionally, S1 comprises two control input channels driven by R2 and R3, respectively. In figure 10.15 the pseudo SDL code of S1 is given.

S2: receives values from R2 via its data input channel and forwards it to AK. Furthermore, a control input channel driven by R3 is provided. Figure 10.16 depicts the pseudo SDL code of S2.

EK: periodically sends current values E_i to the three nodes R1, R2, R3 (see figure 10.11).

AK: periodically transmits the output values A_i to nodes R2 and R3 (see figure 10.17).

DAK: is an auxiliary process implementing the delay occurring on AK. It is subsumed with AK in the component overview in figure 10.10. The SDL pseudo code of DAK is provided in figure 10.18.

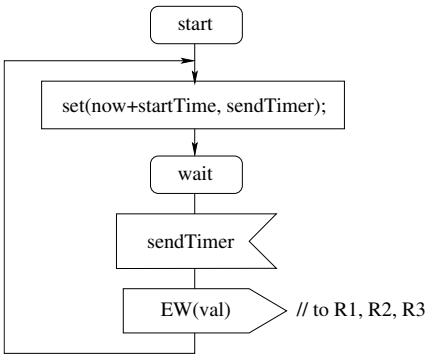


Figure 10.11.: Pseudo SDL Model of EK of the 2-Switch Protocol.

Design Fault. The design fault inserted into the 2-Switch Protocol is a miscalculated deadline. In node R2, `maxTimer` (top left hand side in figure 10.15) is set to expire too late. Thus, the interval in which a value from R1 is expected to be considered on time is stretched. With the modified expiration time it is possible that the value from R1 arrives too late (due to faulty behavior of a component) but is considered on time.

EK and AK (and thus DAK) are assumed to be in the perfection core. Faulty nodes may exhibit arbitrary faulty behavior in the time and value domain. A faulty switch may either omit transfer

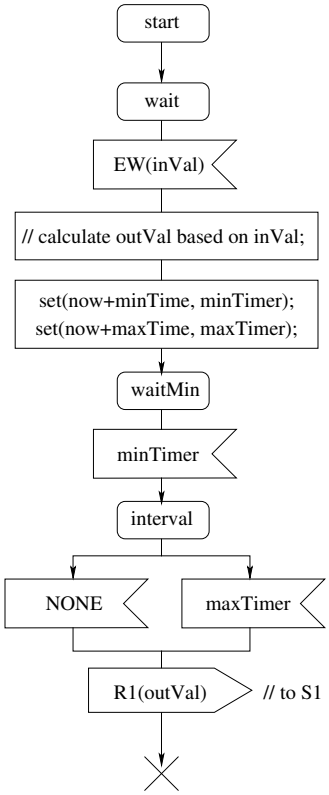


Figure 10.12.: Pseudo SDL Model of R1 of the 2-Switch Protocol.

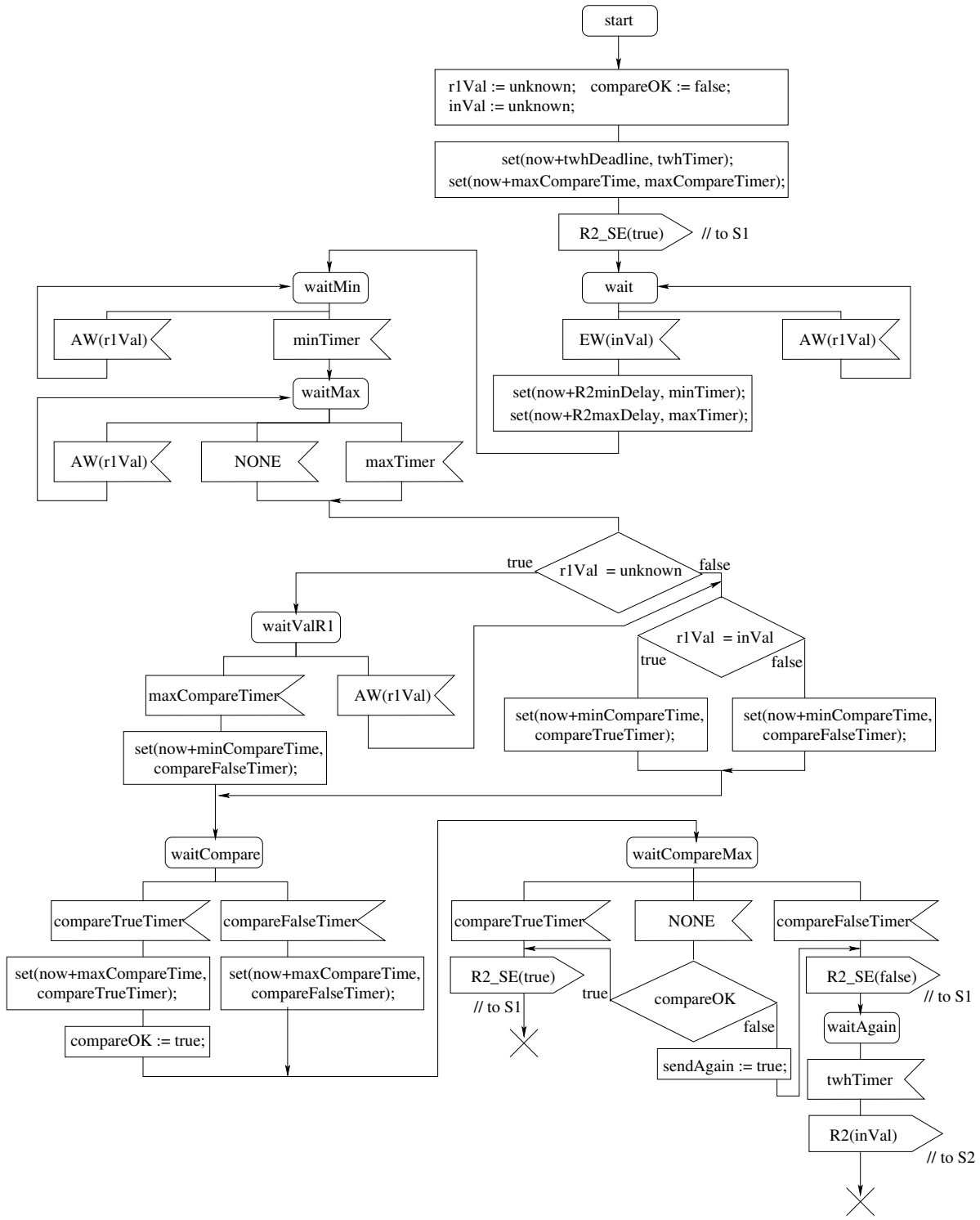


Figure 10.13.: Pseudo SDL Model of R2 of the 2-Switch Protocol.

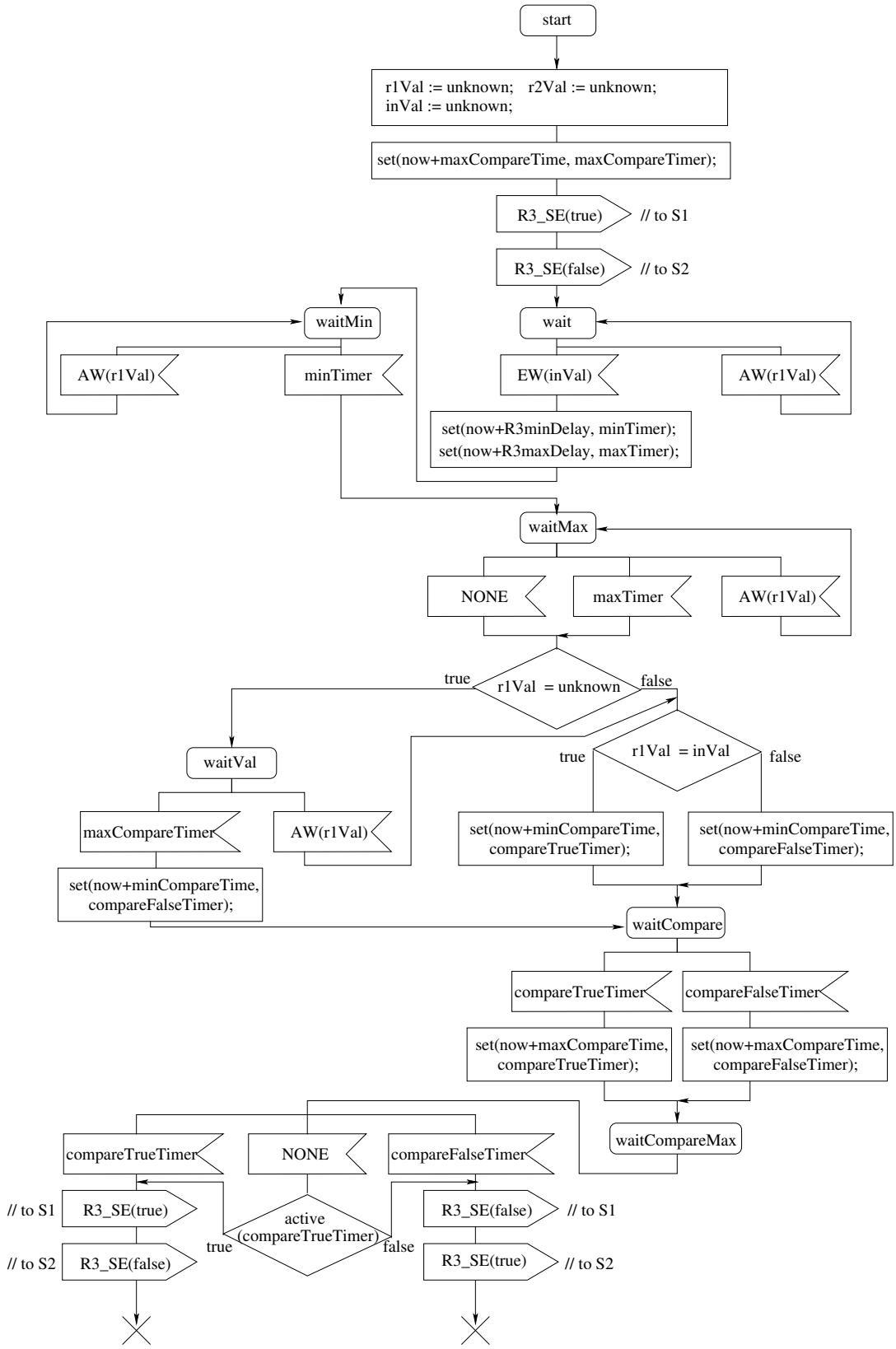


Figure 10.14.: Pseudo SDL Model of R3 of the 2-Switch Protocol.

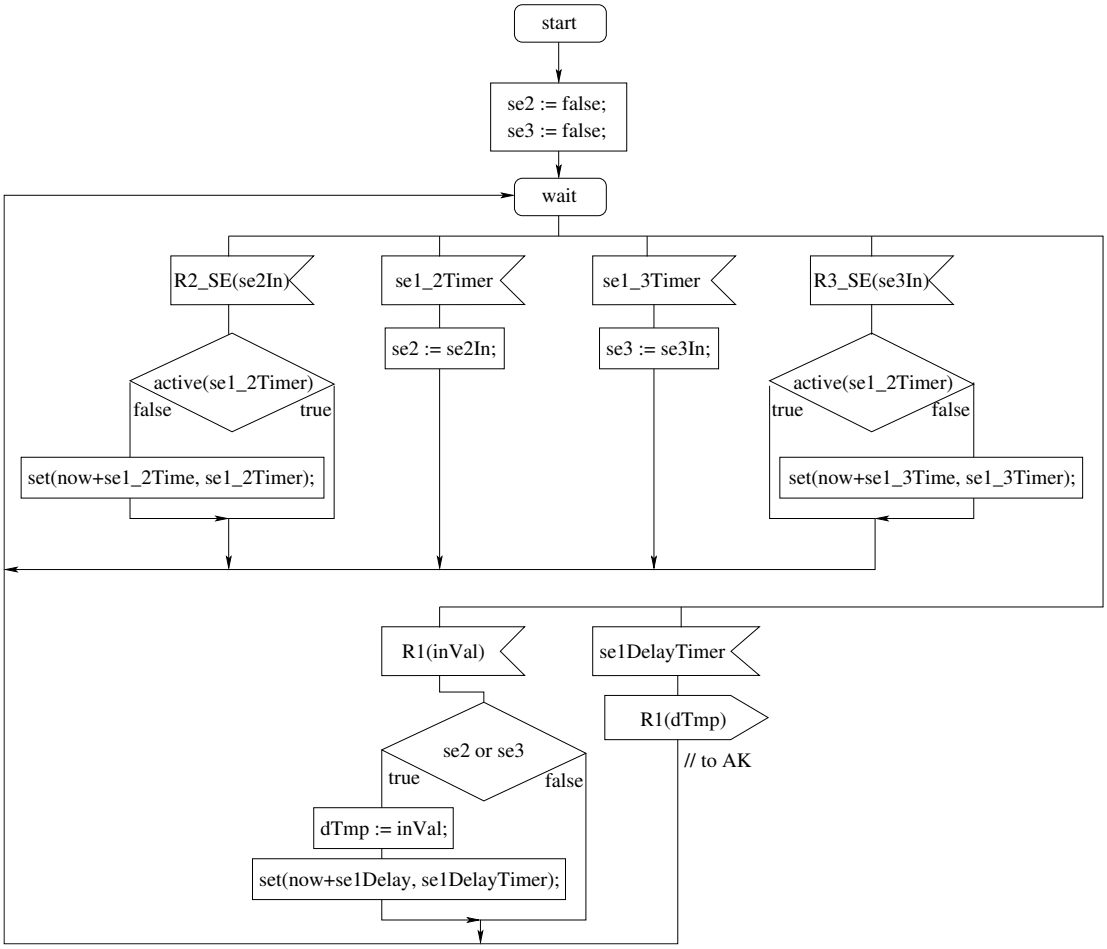


Figure 10.15.: Pseudo SDL Model of S1 of the 2-Switch Protocol.

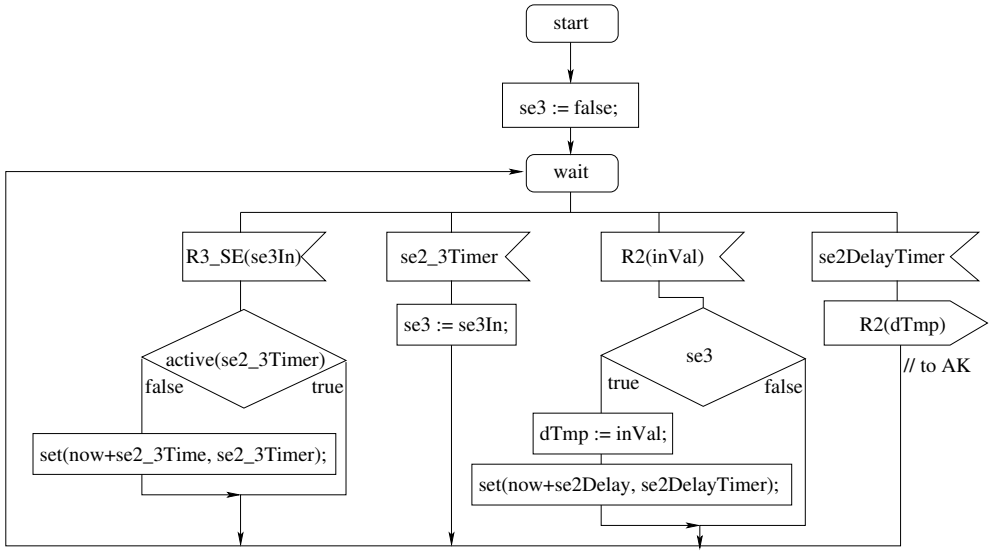


Figure 10.16.: Pseudo SDL Model of S2 of the 2-Switch Protocol.

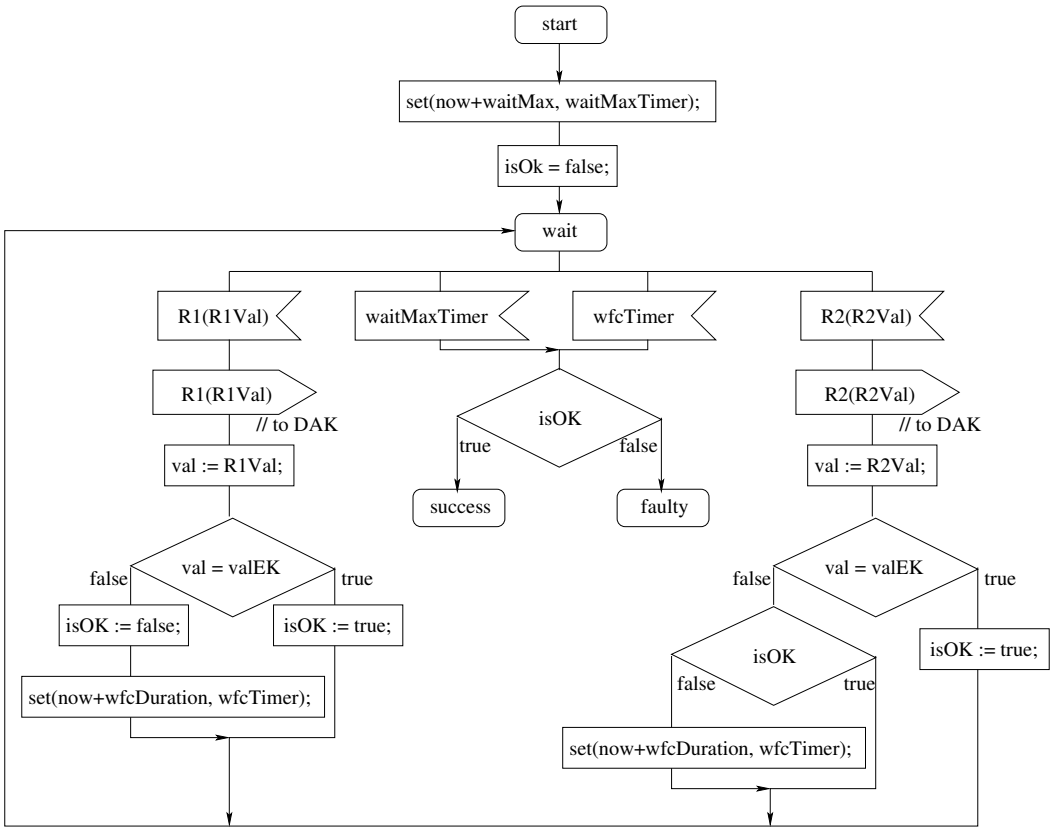


Figure 10.17.: Pseudo SDL Model of AK of the 2-Switch Protocol.

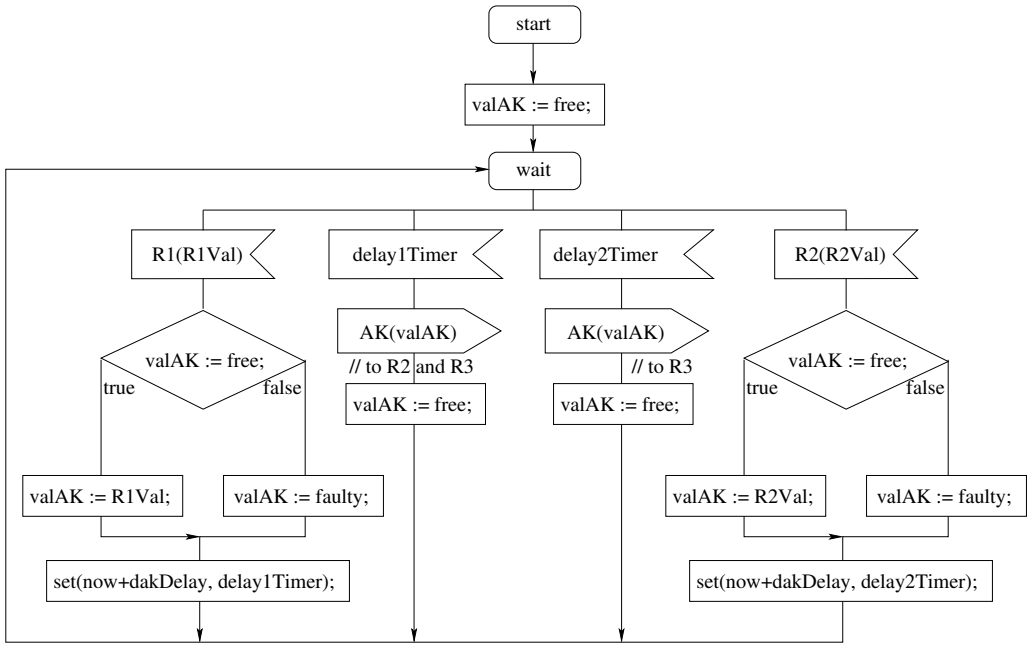


Figure 10.18.: Pseudo SDL Model of DAK of the 2-Switch Protocol.

of a value if it had to be transferred, or transfer a value although it shouldn't. Furthermore, transferred values may be altered (consistently or inconsistently). However, a switch may never output a value if no information is present at its input. Furthermore, faulty switches do not exhibit faults in the time domain.

Experimental Setup. The system is set up as depicted in figure 10.10. The fault-tolerance properties that have to hold are: "A value E_i sent by EK is correctly transmitted by AK (as A_i) at most 80ms after E_i is sent." and "Every third output value may be wrong if 40ms after the wrong value, the correct one is received."

These properties shall hold if the following timing properties are implemented: The input channel EK sends an E_i every 50ms. The transfer of E_i on the input channel takes 8ms as well as the transfer on the output channel AK. Calculation at each node varies between 5ms and 20ms. Decisions require 1ms to 3ms. Switching of S1 and S2 requires 1ms as well as forwarding a signal through a switch.

10.7. FlexRay Protocol

FlexRay is a large protocol for industrial fault-tolerant communication systems for safety-critical automotive applications ([BBB+00, Fle02]). It ensures reliable message transfer in a two-channel environment. An example topology is shown in figure 10.19. Four nodes are connected by a double channel. Channel A is implemented by a bus, channel B contains an active star. Basically, the star broadcasts signals arriving at one branch to its other branches.

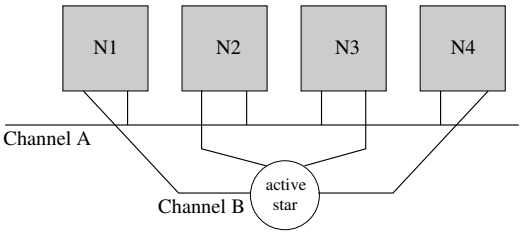


Figure 10.19.: Example Topology of a FlexRay System.

FlexRay is a time-triggered protocol based on rounds. Each round consists of two segments: the static segment and the dynamic segment. The dynamic segment is not considered here as it does not provide any fault-tolerance. The static segment is subdivided into slots. Each slot is assigned to a single node that has exclusive access to the communication channel during this period.

Figure 10.20 gives an overview of the components of a FlexRay system. The *Host* receives data

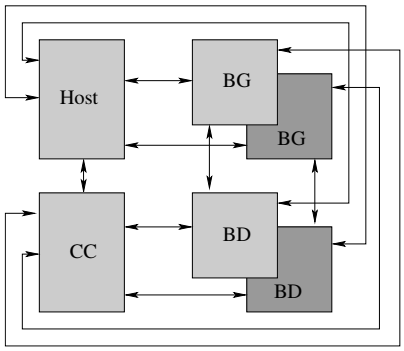


Figure 10.20.: Components of a FlexRay System.

and status information from all other components. It is not part or not of the protocol itself. The *C*ommunication *C*ontroller passes data to the *B*us *D*river and enables or disables the *B*us *G*uardian. The *B*us *D*river forwards the data received from the *CC* to the respective physical layer if the *BG* is open. If a two-channel topology is implemented, two *BD*s are required: one for each channel. Access to each channel may be guarded by a *B*us *G*uardian. The *BG* is an optional element of the FlexRay protocol. It ensures that the *BD* will not transmit on a channel outside its slot.

The FlexRay model is too huge to present the SDL code even in pseudo code. The protocol is based on dozens of parameters that have to be determined very carefully. The design fault of the FlexRay model is based on miscalculation of timing parameters. They are set such that − in very rare cases − it is possible that one node accepts a transmission while the other doesn't.

Experimental Setup. The FlexRay system considered for the experiments consists of three nodes connected through a double channel bus topology. Each component is considered faulty

in turn. Faulty components may exhibit arbitrary behavior resulting in faults in both the time and the value domain.

11. Analysis of Single Fault Region Partial Ordering

This chapter presents the evaluation of the single fault region partial ordering technique SFR-PO as introduced in section 5.1. The evaluation is split into two parts. The analysis of potential performance improvements and an experimental analysis. In the evaluation of the potential, in section 11.1, the results of an analysis run without SFR-PO have been investigated by asking the question "What would have happened, if SFR-PO was applied, and thus only one representative sequence had been applied instead of all?". In the experimental part in section 11.2, example improvements of all of protocols presented in section 10 have been investigated. For each protocol the performance with and without SFR-PO has been compared.

11.1. Analysis of SFR-PO Potential

The analysis of the potential of the SFR-PO approach is based on the FlexRay protocol model (see section 10.7). The model is the most complex of the protocols. With this analysis it should be shown, that applying SFR-PO to huge state spaces results in a significant reduction of unnecessary transitions.

The model has been analyzed by performing an exhaustive state space exploration − restricted by a 48 CPU-hours run-time limit and 1GB of main memory − with the SDT tool. A user-defined rule has been specified describing a violation of the fault-tolerance properties. All paths leading to the violation have been preserved as message sequence charts (MSC [ITU93a]). The other paths have been discarded. During the analysis, 6389 paths have been found leading to the violation.

From these paths, ten pairs of concurrent transitions have been randomly selected. Each pair has been selected satisfying the following conditions :

- **C1:** the 2 transitions are located in different single fault regions;
- **C2:** their order of firing is irrelevant, but both orders can be found in some of the paths generated by SDT.

C1 and C2 ensure that the transitions of each pair would be executed in both sequences if SFR-PO is not applied.

With SFR-PO, the single fault regions are processed sequentially during reachability analysis to achieve partial ordering. Their ordering is irrelevant. Thus, the probabilities for each possible

sequence to be selected as "the" sequence to process the single fault regions are equal. For the ten transition pairs, 44 different execution sequences have been found in the 6389 paths. With SFR-PO only one of these sequences would be executed.

Table 11.1 shows for each of the 44 sequences the number of paths remaining from the 6389, if the respective sequence would have been chosen. The first rows indicate the sequence number, the second rows the number of remaining paths.

Sequence	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
Paths	25	2	11	17	14	38	3	10	8	14	1

Sequence	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
Paths	59	191	10	19	33	22	27	33	579	3273	28

Sequence	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*	*33*
Paths	80	545	544	55	128	68	287	6	25	52	46

Sequence	*34*	*35*	*36*	*37*	*38*	*39*	*40*	*41*	*42*	*43*	*44*
Paths	22	47	3	4	6	35	5	4	3	2	3

Table 11.1.: Analysis of Potential SFR-PO Benefits.

The results show a clear drop in the number of generated paths when applying SFR-PO. The highest number of remaining paths would be generated if sequence 21 was applied. 3273 of the 6389 paths would remain, this is a drop by 48.77%. Applying sequence 11, the perfect result of just one path would be achieved. In the average over the 44 sequences 145.16 paths of the 6389 remain. Thus, a reduction of 97.73% in the average is achieved by applying SFR-PO to even a small number of transition pairs.

11.2. Experimental Analysis of SFR-PO

In the previous section, an example has been discussed to elicit the state space reduction abilities of SFR-PO. The analysis in this section focuses on experiments with the implemented SFR-PO mechanism in order to substantiate the discovered benefits. For all of the protocols presented throughout section 10, a reachability analysis without applying SFR-PO has been performed, followed by a reachability analysis applying SFR-PO. The algorithm for the analysis is the exhaustive algorithm as presented in section 3.2.1 to maintain comparability with the analysis of the potential (section 11.1).

Experimental Setup. The setup is equivalent to the one of section 11.1. However, all protocols presented throughout section 10 are included in the analysis. Furthermore, not a single global

state is picked per protocol, but 10 global states are chosen. Thus, an average reduction factor can be determined. In order to be selected, a global state had to fulfill the following criteria:

- It had to be reached during the exhaustive analysis without applying SFR-PO. Otherwise, no basis for comparison would be available;
- at least 100 different paths to the state had to be generated during the non-SFR-PO run to provide a minimum basis for comparison. In other words: no reliable reduction factors can be assumed if too few paths are considered;
- conditions C1 and C2 can be fulfilled as for the analysis of the potential (see section 11.1).

For each of the selected global states the approach is the same as in the analysis of the potential (section 11.1). Ten transition pairs fulfilling conditions C1 and C2 are selected.

Two exploration runs were performed per selected global state and transition pair: The first run not applying SFR-PO, the second one applying SFR-PO. The number of (different) paths to this state has been compared as in the analysis of section 11.1.

Table 11.2 summarizes the results. It shows for each protocol (see first row) the number of generated paths averaged over the ten runs. The number of paths generated without SFR-PO are displayed in the second row. The third row provides the highest number of remaining paths when applying SFR-PO. In other words: not the complete numbers as in table 11.1 are shown, but only the worst case is selected. Thus, the last row contains the average minimum reduction factor achieved through application of SFR-PO.

Protocol	**PP**	**RBA1**	**DBA1**	**SM**	**VETO**	**2SP**	**FlexRay**
no SFR-PO	1527.3	4392.8	209.3	1874.0	5057.2	3605.8	6469.6
SFR-PO	235.1	629.9	73.6	788.8	2278.8	1634.4	2946.6
Reduction	84.61%	85.66%	64.34%	57.91%	54.94%	54.67%	54.45%

Table 11.2.: Experimental Analysis of SFR-PO Benefits.

The results of the experimental analysis substantiate the potential benefits. For all of the protocols applying SFR-PO yields a reduction of more than 50%. For the pendulum protocol and the RBA1 protocol, the reduction is even about 85%. Thus, it can be concluded that application of SFR-PO yields a substantial benefit towards skipping unnecessary transition execution sequences. Thereby allowing to explore a larger portion of the state space within potential run-time and main memory limits.

SFR-PO will be applied to all further experiments in subsequent sections.

12. Analysis of the H-RAFT Algorithm

Chapter 12 contains the evaluation of the H-RAFT algorithm. The experiments are composed incrementally. First, the general reduction techniques are evaluated in section 12.1. They comprise the width restriction of the state space, the investigation of suitable depth factors (introduced in section 6.2), the restriction of spontaneous transitions in processes representing faulty components and the elimination of signal-consumption-only transitions (introduced in section 6.3.1). For all of these experiments, the SFR-PO technique is applied. The best settings found for the general reduction techniques are applied to all further experiments. In section 12.2, the performance of the H-RAFT algorithm with different settings of the input weights (section 6.3.1) is evaluated. Throughout this series of experiments, an assignment of weights to the different input elements is derived that yields the best results summarized over all of the investigated protocols. The impact of the action weights (section 6.3.2) on finding fault-tolerance violations is investigated in section 12.3. Furthermore, combinations of action and input weights are considered there. The overall goal of the evaluation is to determine and substantiate the best weight settings for all SDL elements.

Applying these settings to *any* fault-tolerant communication protocol should clearly improve the chances of finding a fault-tolerance-property violation. This assumption will be substantiated by checking it against all of the modeled protocols described in section 10. A comparison of the performance of the H-RAFT algorithm with the Close-To-Failure algorithm (as presented in section 7) and the general algorithms (section 3.2) is provided in chapter 14.

The basis for comparing the algorithms and settings are the two criteria formulated in section 1.1 (page 5):

1. they find fault-tolerance violations that have not been detected by general algorithms for reachability analysis and/or
2. they find fault-tolerance violations faster than the general algorithms. In other words: less transitions had to be performed.

General Experimental Setup. The experiments described throughout this section follow the limitations motivated in section 9. They are limited to a run-time of 48 cpu-hours and the available main memory is always 1GB.

12.1. General Reduction Techniques

The purpose of this series of experiments is to investigate different combinations of width restrictions and depth factors as introduced in section 6.2 (pages 60 and 62) resulting in the state-weight

equation 6.4 (page 62). Furthermore, different decrease factors *nDec* (equation 6.7, page 67) for spontaneous transitions in processes representing faulty components are investigated. Finally, the effects of eliminating signal-consumption-only transitions (equation 6.9, page 67) are evaluated. In order to provide substantial results, all combinations of the different parameter values have been investigated for each of the protocols described in chapter 10. Transition weights are not considered in this series of experiments − they are subject to chapters 12.2 and 12.3.

12.1.1. Experimental Setup

The following paragraphs discuss the co-domain selection for each of the investigated parameters.

Width Restriction. This parameter allows for comparison of different widths of the reachability graph. In other words: to limit the number of global states allowed on each front respectively level of the graph (see section 6.2). Width restriction has been introduced to enforce exploration of deeper parts of the reachability graph. Five values have been selected ranging from 100 to 1000 allowed states on each front: $\boldsymbol{width \in \{100,250,500,750,1000\}}$. Restricting the width below 100 is not advisable as this would supposedly mask out too much of the reachability graph. Widths of more than 1000 are also not investigated. It is unlikely that much more than 1000 states are active concurrently. Even if this was the case, those additional states are not very likely to be selected for further analysis as their global state weights are smaller than those of the first 1000 states. This parameter range selection has also been observed to represent reasonable choice during the experiments.

Depth Factor. The depth factor has been introduced to prevent outgrowing of a single subtree (see section 6.2, page 62). It has been included in the global state weight *wState* by introduction of a function *df*:

$$wState(s_i) = \max_{tr_j \in activeTRset(s_i)} \{wTrans(tr_j)\} - df(depth(s_i))$$

(equation 6.3, page 59).

Four different functions have been considered for this factor here:

- $df_1(\mathbf{depth(s_i)}) = \mathbf{0}$: No depth factor is provided;
- $df_2(\mathbf{depth(s_i)}) = \mathbf{depth(s_i)}$: The depth of the global state represents the reduction factor;
- $df_3(\mathbf{depth(s_i)}) = \mathbf{0.01 \cdot depth(s_i)}$: As for df_2, the depth of the global state is the basis for the reduction factor. The factor of 0.01 indicates that the resulting value, and thus the influence, of df_3 with respect to the total global state weight is reduced.
- $df_4(\mathbf{depth(s_i)}) = \sqrt{\mathbf{depth(s_i)}}$: While the previous functions are linearly dependent on the global state's depth, df_4 reduces the weight more slowly, the deeper the states are located.

Many other *df*-functions are conceivable. df_1 to df_4 have been selected as they mark representative distributions. After investigating the effects of these four functions, the necessity for investigating further functions was not given anymore.

Spontaneous Transitions in Faulty Components. In section 6.3, decreasing weights for spontaneous transitions in faulty components have been discussed. The idea behind this approach is based on the assumption that spontaneous transitions in faulty components represent certain (in most cases faulty) behaviors which may occur repeatedly. With the "*any output at any time*" fault-model, these transitions usually model sending a signal to an adjacent component at an undetermined time. This can be repeated an arbitrary number of times. When decreasing the weight of each transition every time it has fired, the number of repetitions is decreased - not the points in time when they may fire. Calculation of the current weight of each spontaneous transition in a faulty process is provided by: $wInput(input(tr_j)) = wNone \cdot nDec^{fired(tr_j)}$, where *fired(tr_j)* denotes how many times transition tr_j has fired before (equation 6.7, page 67).

Although this weight is defined as a transition input weight, it is considered in this series of experiments to investigate their influence more thoroughly. If considered along with the other input element weights, it is not possible to concentrate on the impact of these rather special transition weights.

Three different decrease factors have been considered: $\boldsymbol{nDec \in \{1.0,\ 0.5,\ 0.1\}}$. If *nDec=1.0*, no decrease factor is employed. By setting *nDec=0.5*, the transition weight is halved every time the transition fires. *nDec=0.1* has been selected to investigate the effects if each of those transitions will fire only a very limited number of times, as their weight is decreased very rapidly.

Signal-Consumption-Only Transitions. The last parameter of this experiment series focuses on the benefits of eliminating signal-consumption-only (SCO) transitions that do not support the progress of the reachability analysis (page 67). This is a binary parameter. Either SCO transitions are ignored (*SCO="ignore"*) or they are treated as normal transitions (*SCO="consider"*).

Experimental Setup. All combinations of the four parameters are investigated and the results are denoted in tables 12.1 to 12.6. Tables 12.1 to 12.3 show the results when SCO transitions are considered as normal transitions. Tables 12.4 to 12.6 represent those experiments where SCO transitions are eliminated. The three tables per group contain the results for the three *nDec* factors respectively. Each table depicts the combinations of the width restrictions and the depth factors. In the cells, the number of the protocols for which a fault-tolerance-property violation has been found − with the parameter combination under consideration − is denoted. The higher the number in a cell, the more promising a parameter combination is for evaluation of fault-tolerance protocols.

12.1.2. Results

Tables 12.1 to 12.6 provide a summarized overview of the performance of each parameter combination. Each of the tables displays the utilized depth factors in the columns and the applied width restrictions in the rows. The first three tables (12.1 to 12.3) display the results for SCO="consider" and the different values for *nDec*. The other three tables provide the results for SCO="ignore".

The experiments have been carried out for each of the modeled protocols. Every time a fault-tolerance-property violation has been found, it has been marked in the respective cell representing the used parameter combination. Thus, the numbers in the tables show for how many of

the protocols a violation has been found with the respective parameter combination. Thereby, the first criterion for the evaluation of the novel techniques (see page 123) is evaluated. In other words: combinations with a higher number of hits are more suitable for finding fault-tolerance violations in general.

The secondary criterion, the number of transitions until a violation is detected, is only taken into account if the number of hits is equal and no clear tendency is visible in favor of any of the parameter values.

By this kind of display, a summary over the protocols is provided. The distribution of the hits for the different protocols is broken down in table 12.12. Table 12.12 depicts for each of the protocols the property violations found in each of the first six tables. In other words: the number of hits per table is counted for each protocol. Thereby, it can be evaluated whether certain combinations are suitable for all protocols, or only for a subset of protocols.

depth factor → ***width*** ↓	$\mathbf{df_1}$	$\mathbf{df_2}$	$\mathbf{df_3}$	$\mathbf{df_4}$
1000	2	2	2	2
750	3	2	2	3
500	3	3	3	2
250	4	2	2	4
100	4	2	2	4

Table 12.1.: Width and Depth Restriction, nDec=1.0, SCO=”consider”.

depth factor → ***width*** ↓	$\mathbf{df_1}$	$\mathbf{df_2}$	$\mathbf{df_3}$	$\mathbf{df_4}$
1000	2	2	3	2
750	3	3	2	3
500	3	4	4	3
250	4	2	3	5
100	5	2	2	4

Table 12.2.: Width and Depth Restriction, nDec=0.5, SCO=”consider”.

depth factor → ***width*** ↓	$\mathbf{df_1}$	$\mathbf{df_2}$	$\mathbf{df_3}$	$\mathbf{df_4}$
1000	3	3	3	3
750	3	3	3	3
500	3	3	4	3
250	4	2	4	4
100	5	2	2	5

Table 12.3.: Width and Depth Restriction, nDec=0.1, SCO="consider".

depth factor → ***width*** ↓	$\mathbf{df_1}$	$\mathbf{df_2}$	$\mathbf{df_3}$	$\mathbf{df_4}$
1000	2	2	2	2
750	3	3	3	3
500	2	3	3	2
250	3	3	3	2
100	4	2	2	3

Table 12.4.: Width and Depth Restriction, nDec=1.0, SCO="ignore".

12.1.3. Discussion of the Results

Tables 12.1 to 12.6 allow for evaluation of each parameter separately and for evaluation of parameter-combinations. First, a discussion of each parameter is given, then the combinations are discussed. The only exception is the distinction between considering SCOs and ignoring them. This distinction is made for each of the other parameters.

Width Restriction. Table 12.7 provides a summary of the experiments focusing on the weight factor only. Each cell contains the sum over the rows of the indicated table. The last two rows contain the sum over tables 12.1 to 12.3 resp. 12.4 to 12.6. In other words, the summary of hits when considering SCOs and ignoring them.

Only few general statements on the most suitable width can be made when looking at the sums only. Obviously, the performance is worst for a width restriction of 1000. When ignoring SCOs, a width of 100 yields the most hits. When considering SCOs, a width of 250 seems suitable,

depth factor → ***width*** ↓	$\mathbf{df_1}$	$\mathbf{df_2}$	$\mathbf{df_3}$	$\mathbf{df_4}$
1000	2	2	2	2
750	3	3	3	3
500	2	4	3	2
250	2	4	4	2
100	5	3	3	5

Table 12.5.: Width and Depth Restriction, nDec=0.5, SCO="ignore".

depth factor → ***width*** ↓	$\mathbf{df_1}$	$\mathbf{df_2}$	$\mathbf{df_3}$	$\mathbf{df_4}$
1000	2	2	2	2
750	3	3	3	3
500	2	4	3	2
250	2	4	4	2
100	5	3	3	5

Table 12.6.: Width and Depth Restriction, nDec=0.1, SCO="ignore".

while a width restriction to 100 or 500 does not perform much worse either. Thus, it becomes evident, that width restriction without taking further information on the other parameters into account is not conclusive. These combinations are discussed in later paragraphs.

Depth Factor. Four different functions (df_1 to df_4) taking the depth of the global state into account have been investigated. Similar to table 12.7, table 12.8 provides a summarized overview of the different hits. Each cell contains the sum of the respective columns. The results show a clear preference for df_1 and df_4 when considering SCOs. When ignoring SCOs, df_2 slightly outperform the other three functions. Again, no clear statement about this parameter independent of the other parameters is possible.

Spontaneous Transitions in Faulty Components. Table 12.9 provides a summary of the combination of *nDec* and *SCO*. The cells contain the summarized number of hits taken from tables

width → ***table*** ↓	100	250	500	750	1000
12.1	12	12	11	10	8
12.2	13	14	14	11	9
12.3	14	14	13	12	12
12.4	11	11	10	12	8
12.5	16	12	11	12	8
12.6	16	12	11	12	8
$\sum_c$	39	40	38	33	29
$\sum_i$	41	35	32	36	24

Table 12.7.: Summary of Width Experiments.

depth factor → ***table*** ↓	df_1	df_2	df_3	df_4
12.1	16	11	11	15
12.2	17	13	14	17
12.3	18	13	16	18
12.4	14	13	13	12
12.5	14	16	15	14
12.6	14	16	15	14
$\sum_c$	51	37	41	50
$\sum_i$	42	45	43	40

Table 12.8.: Summary of Depth Factor Experiments.

12.1 to 12.6. Decreasing the weight of spontaneous transitions of faulty processes each time they fired, increases the chances of finding fault-tolerance violations. Not decreasing the weights (nDec=1.0) results in far the least hits. Considering SCOs in combination with fast decreasing weights (nDec=0.1) provides the highest number of hits. When ignoring SCOs, both nDec=0.1 and nDec=0.5 provide better results than nDec=1.0. The reason for the correlation between SCOs and fast decreasing can be seen in that many transitions executed in faulty processes can be assumed to be SCOs. nDec is thus the first parameter yielding a clear value preference

nDec→ ***SCO*** ↓	**0.1**	**0.5**	**1.0**
consider	65	61	53
ignore	59	59	52

Table 12.9.: Summary of nDec Experiments.

independent of the other parameters.

Parameter Combinations. While the results for *width* and *depth* alone are not very clear, the selection of *nDec=0.1* is obvious. The results for the parameter-combinations where nDec=0.1 are shown in tables 12.3 and 12.6 only. Table 12.10 contains the sum of the two tables under consideration. Thus, both width and depth can be distinguished while the SCO parameter need not be considered in the table. The last column provides the total number of hits for the

depth factor → ***width*** ↓	$\mathbf{df_1}$	$\mathbf{df_2}$	$\mathbf{df_3}$	$\mathbf{df_4}$	**SUM**
1000	5	5	5	5	**20**
750	6	6	6	6	**24**
500	5	7	7	5	**24**
250	6	6	8	6	**26**
100	10	5	5	10	**30**
SUM	**32**	**29**	**31**	**32**	

Table 12.10.: Width and Depth Restriction Summary for nDec=0.1.

respective width. The bottom row contains the summarized hits of each depth function. Now, the tendency for selecting a width restriction is obvious. The tighter the restriction, the more hits are counted. This has even been observed for the huge FlexRay model where a width of 100 yields the most hits. Thus the width parameter can be set to 100. Then, the selection of the suitable depth function is limited to two choices. Either df_1 is applied or df_4. As the performance is equal for both functions − even when distinguishing between considering SCOs and ignoring them − $df_1(\mathbf{depth}(\mathbf{s_i})) = \mathbf{0}$ is selected. This function represents applying no depth-factor at all. Thus, this calculation is saved during execution.

Signal-Consumption-Only Transitions. Comparison of the two possibilities of setting SCOs can be provided by summing up the number of hits in the first three tables for SCO="consider" and the last three tables for SCO="ignore". Table 12.11 provides these numbers. Looking at

SCO	**consider**	**ignore**
hits	179	170

Table 12.11.: Summary of SCO Experiments.

the summarized values only, considering SCOs clearly outperforms ignoring them. Looking at table 12.9 again, considering SCOs yields better results as well. Thus, the last parameter can be fixed.

SCO →	*consider*			*ignore*		
table → *protocol* ↓	12.1	12.2	12.3	12.4	12.5	12.6
PP	20	20	20	20	20	20
FX	0	7	8	0	7	7
RBA1	0	0	0	2	2	2
DBA1	20	20	20	20	20	20
SM	7	8	9	0	0	0
2SP	2	2	3	8	8	8
VETO	4	4	5	2	2	2
Sum	53	61	65	52	59	59

Table 12.12.: Detected Property Violations for Each Protocol.

Protocol Table Discussion. So far, a clear tendency towards considering SCOs has been visible. However, this result becomes questionable comparing the performance for the different protocols. Table 12.12 provides the number of hits of the respective protocol (rows) for each table (columns). For the small protocols PP and DBA1, all parameter settings yield a hit. For most of the other protocols a clear distinction between the two series of experiments concerning SCOs can be seen. For the RBA1 and the 2SP protocol, a clear tendency towards ignoring SCOs is observed. For the SM and the VETO protocol on the other hand, considering SCOs is favorable. These contradictory results can be "blamed" on differences in the modeling techniques. The faulty component of RBA1 and 2SP are modeled such that faulty behavior may occur at any time without any restriction in the time domain. Many transitions indicating faulty behavior are unnecessary and do not lead to changes in the system behavior. In other words: during execution of the RBA1 resp. the 2SP model, a huge amount of Signal-Consumption-Only transitions occur. In the other protocols, some restrictions on the points of time where faulty behavior may occur are made such that SCOs are eliminated during modeling time already. For the FlexRay protocol, no significant difference between considering and ignoring SCOs is visible. These results suggest not fixing SCO yet but observing it throughout further experiments.

Summary. Through an incremental approach values for three of the four parameters could be fixed. The best performance can be expected for the following setting:

- **width** = 100;
- **df** = 0;
- **nDec** = 0.1.

Whether SCOs should be considered or ignored could not be determined yet. Thus, both cases will be observed throughout the experiments of the following sections. All in all, it can be claimed that width restriction and decreasing the weight of transitions in faulty components each time they fired yields a significant performance gain. Reducing the weight of states located deeper in the graph in order to prevent outgrowing of a subtree did not prove advantageous.

12.2. Input Weights

In this section experiments evaluating different transition input weights (section 6.3) are discussed. The goal is to find an input weight combination, for all seven input elements, that can be applied to all models of fault-tolerant protocols to increase the chances of finding fault-tolerance-property violations.

So far, the best-performing values for the parameters "width", "depth factor", and "nDec" have been determined. These values will be used for the experiments determining well-suited input element weights. The "width" is restricted to 100 global states per front and the depth factor is set to $df(depth(s_i)) = 0$. The decrease factor $nDec$ for spontaneous transitions in faulty processes is set to $nDec = 0.1$. Since the results for ignoring or considering Signal-Consumption-Only transitions have been ambiguous, both variants are considered in the experiments of this section observing whether a clear tendency towards using or ignoring SCOs can be made out.

Run-time and main memory are again limited to 48 cpu-hours and 1GB, respectively.

For each of the static weights of the seven input elements, seven different values, w_0 to w_6, are assumed: zero ($w_0 = 0$) and 6 weights (w_1 to w_6) greater than zero, where $w_0 < w_1 < w_2 < w_3 < w_4 < w_5 < w_6$. The w_0 to w_6 are also the weights of the transitions. Thereby, all of the possible tendencies among the input elements are investigated. For each input element, any of the w_i may be selected independent of the selection for the other input elements. As each transition has exactly one input element, this selection of available weights allows for considering all combinations. In other words: all weight combinations relative to each other may be investigated. Thus, the maximum number of experimental runs is 7^7 (= 823,543) if all combinations are evaluated.

Finding an optimal weight combination is done in a two-step process. First, the optimal combination for a single one of the modeled protocols is determined. Then, in a second step, the plausibility of this weight combination is checked. Determination of the optimal weight combination is achieved through experimental runs for all weight combinations. A combination is considered "optimal", if a fault-tolerance-property violation has been found (with a minimum number of transitions).

In order to substantiate such a promising combination of the input weight values, the neighborhood of the mesh point representing the combination of the weight values with the highest percentage of fault-tolerance-violating paths found is investigated. The seemingly optimal weight combination may be coincidental. With only a slight change in the ordering of the weights, unsuitable weight combinations may emerge. This is also investigated by considering the complete neighborhood of the supposed optimum. If the results in the neighborhood are close-to-optimal, the optimum weight combination can be recommended for investigation of any model. In other words, the robustness of the "optimal" weight combination is evaluated.

An example for the neighborhood of an optimum in two dimensions is shown in figure 12.1. The inner black dot marks the combination of the optimum weight values. In this case, w_1 for the input element "timestep" and w_2 for the input element "spontaneous transition" represent the optimum weight combination. The circle comprises the neighbors of that point. By "hits"

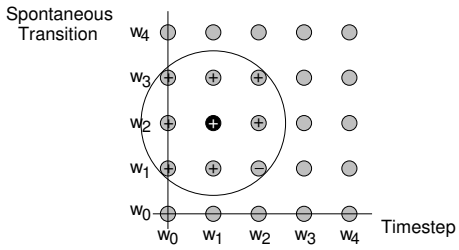


Figure 12.1.: Neighborhood of Optimum Combination of Weights (2D-Example).

(marked by a "+" in figure 12.1), the mesh points representing the weight combinations of an experimental run that has found a fault-tolerance-violating path are denoted. The other mesh points are referred to as "misses" (marked by a "−"). By counting the hits and misses within the neighborhood the quality of the results can be measured.

In order to find an optimal weight assignment to the seven input elements, the best solution would be to check all 7^7 (= 823,543) possible combinations. With a maximum run-time of 48 CPU-hours per experimental run, this may result in 1,647,086 days (≈ 4512 years) for a single protocol. This is not feasible. Thus, an incremental approach is pursued to determine (at least a local) optimum weight assignment for the seven input elements. One of the implemented models contains only five of the seven input elements, some contain 6 different input elements and only a fraction requires all seven. Table 12.13 gives an overview. The entry in the first column of

Number of Input Elements	**Input Elements**	**Protocols**
IE 5	"timer expired", "none", "signal with parameters", "timestep", "timer ready"	PP
IE 6	+ "timer array expired"	SM, 2SP, RBA1, DBA1
IE 7	+ "signal without parameters"	VETO, FX

Table 12.13.: *I*nput *E*lements of the Models.

the table denotes the name of the set of the protocols using five, six, or seven input elements, respectively. The protocols contained in each set are listed in column "Protocols". In column "Input Elements", the respective input elements are named. "+" means "those of the previous row plus the one mentioned".

The incremental approach is build up as follows: First, the optimum weight combination $IE5_{OPTIMUM}$ for the five input elements is determined for the IE5-protocol. The result is substantiated by investigation of the neighborhood of that determined optimum. In section 12.2.1, this first series of experiments is provided.

The next step is to determine the (local) optimum for the six input elements of the IE6-protocols starting with $IE5_{OPTIMUM}$ and adding the sixth dimension: For the sixth element, the weight is varied such that all combinations relative to all of the five other weights are considered. The weights of the five input elements are not changed. As an illustrative example, let the five already determined weights be (*timer expired, none, signal with parameters, timestep, timer ready*) =

$(w_0, w_1, w_2, w_1, w_0)$. Then seven experiments are required to determine a (local) optimum, where the weight w_{tae} for *timer array expired* (as introduced in section 6.3.1, page 64) is set such:

- $w_{tae} < w_0$;
- $w_{tae} = w_0$;
- $w_0 < w_{tae} < w_1$;
- $w_{tae} = w_1$;
- $w_1 < w_{tae} < w_2$;
- $w_{tae} = w_2$;
- $w_{tae} > w_2$;

The weight combination performing best will be referred to as $IE6_{OPTIMUM}$. $IE6_{OPTIMUM}$ is possibly a local optimum as not all weight combinations have been investigated. This is not a drawback, if performance is increased nevertheless. Again, the plausibility of the optimum is substantiated by investigation of its neighborhood. The analysis is provided in section 12.2.2. Finally, $IE7_{OPTIMUM}$ is determined likewise in section 12.2.3, including analysis of the neighborhood.

12.2.1. Five Input Elements

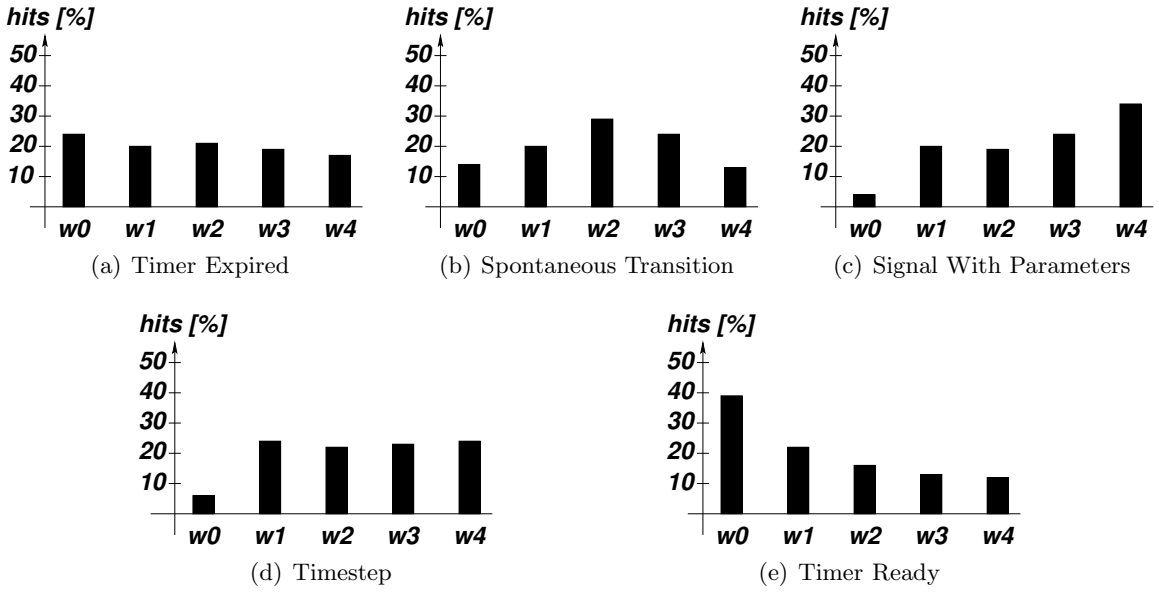
The first step of determining a suitable weight combination for five input elements is to determine one or several optimal combinations referred to as $IE5_{CENTER}$. This (seemingly) optimal combination(s) is/are then substantiated by investigation of its/their neighborhood(s). The term *CENTER* indicates the center point(s) of the investigated neighborhood.

By first investigating five input elements only, the number of experimental runs is reduced to $5^5 = 3125$. As the pendulum protocol (section 10.1, page 97) is the only one requiring five input elements only, its model will serve for determining $IE5_{CENTER}$.

IE5-Center.

The results of this first set of experimental runs are summarized in figure 12.2 (SCO="consider") and figure 12.3 (SCO="ignore"). The figures show − for each input element and each possible weight − the percentage of experiments leading to a fault-tolerance-violating path if the respective weight value w_i (i=0 to 4) for the displayed input element ("timer expired", "spontaneous transition", "signal with parameters", "timestep", "timer ready") has been set. The remaining weight combinations are averaged. In other words: The average over a "slice" of a 5-dimensional matrix is displayed.

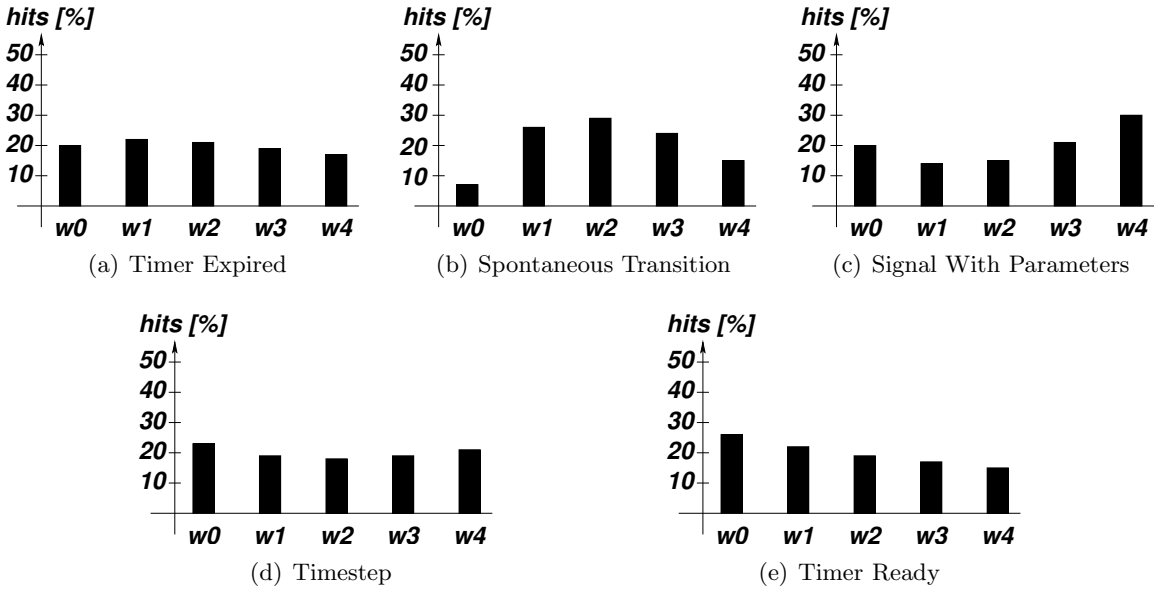
Selection of the optimal weight combination is accomplished by selecting for each input element the weight with the highest percentage of found violations. If Signal-Consumption-Only transitions are fully considered (figure 12.2), the results for the input element "timer expired" show



(a) Timer Expired (b) Spontaneous Transition (c) Signal With Parameters

(d) Timestep (e) Timer Ready

Figure 12.2.: IE5$_{CENTER}$, SCO="consider".



(a) Timer Expired (b) Spontaneous Transition (c) Signal With Parameters

(d) Timestep (e) Timer Ready

Figure 12.3.: IE5$_{CENTER}$, SCO="ignore".

a slight preference for a small weight. The best performance has been observed for a weight of zero. Through a medium value for the weight of spontaneous transitions, the highest rate of finding a fault-tolerance violation is achieved. In contrast, "signals with parameters" yields the best chances of discovering a fault-tolerance violation when assigned a high weight. Apart from setting the weight value of the transition input "timestep" to zero, resulting in a very low success rate, the performance is observed to be almost equally distributed among the different weights. For the "timer ready" input element, a weight of zero shows by far the best chances of finding a fault-tolerance-violating path.

The results if Signal-Consumption-Only transitions are ignored are similar. For the "timer expired" transition input, chances of finding a fault-tolerance-violating path are almost equally distributed among the weights. A small value greater than zero indicates slightly higher chances for encountering such path. The difference between the performances when considering Signal-Consumption-Only transitions (figure 12.2(a)) and ignoring them (figure 12.3(a)) is marginal. Both show a slight preference for small weights.

Comparing the weight distributions of the "spontaneous transitions" in the two experiments (figures 12.2(b) and 12.3(b)), the tendencies are equal. Most violating paths have been found at a medium weight.

Considering "signals with parameters" as inputs, the results of the two experiments differ in one case. Assigning a weight of zero shows unquestionably the lowest chances of finding a fault-tolerance-violating path when Signal-Consumption-Only transitions are allowed (figure 12.2(c)). However, eliminating those transitions (figure 12.3(c)), a weight of zero leads to a mid-range performance. Nevertheless, assigning the highest weight to "signals with parameters" provides the best chances in both experiments.

Similar to the input "signals with parameters", the "timestep" input element shows an increase from below 10% to over 20% for a weight of zero, when eliminating signal-consumption-only transitions (figures 12.2(d) and 12.3(d)). The other weight values are almost equally distributed.

The tendencies with respect to the "timer ready" input did not change with elimination of Signal-Consumption-Only transitions. Lower weights still lead to detection of the most fault-tolerance-violating paths (figures 12.2(e) and 12.3(e)).

For the transition input "timestep", two optimum weight values (w_1 and w_4) have been observed for SCO="consider". The optimal weight assignments ("timer expired", "spontaneous transition", "signal with parameters", "timestep", "timer ready") are thus:

IE5$_{CENTER\_1}$**, SCO="consider" :** (w_0, w_2, w_4, w_1, w_0)

IE5$_{CENTER\_2}$**, SCO="consider" :** (w_0, w_2, w_4, w_4, w_0)

IE5$_{CENTER}$**, SCO="ignore" :** (w_1, w_2, w_4, w_0, w_0)

IE5-Neighborhood.

For SCO="consider", two five-dimensional neighborhoods have to be investigated to ensure the plausibility of the weight combinations. In both cases the center-point (IE5$_{CENTER_1}$, respec-

tively $IE5_{CENTER\_2}$) has been a "hit". For the neighborhood surrounding $IE5_{CENTER\_1}$, the hit ratio was 70.37%. For the neighborhood of $IE5_{CENTER\_2}$, the hit ratio was even 95.24%.

Investigation of the neighborhood around the optimum weight combination $IE5_{CENTER}$, with SCO="ignore" yields a hit-ratio of 88.98 percent.

The hit-ratios prove that the optimum weight combinations $IE5_{CENTER\_1}$ and − even more − $IE5_{CENTER\_2}$ for SCO="consider" and $IE5_{CENTER}$ for SCO="ignore" are indeed suitable for drastically increasing the chances of finding a fault-tolerance violation.

Comparison of the hit-ratios when considering and ignoring Signal-Consumption-Only transitions shows, again, no significant difference between the two approaches. Both variants will be observed in further experiments for six and seven input elements possibly revealing a clear tendency towards one of the settings.

Conclusion of IE5 Experiments.

Since $IE5_{CENTER\_2}$ yields a higher hit ratio in the neighborhood than $IE5_{CENTER\_1}$ for SCO = "consider", $IE5_{OPTIMUM_c}$ and $IE5_{OPTIMUM_i}$ (for SCO="ignore") are set to:

$$\mathbf{IE5}_{OPTIMUM\_c} = \mathbf{IE5}_{CENTER\_2}$$

$$\mathbf{IE5}_{OPTIMUM\_i} = \mathbf{IE5}_{CENTER}$$

These two "optima" are used as baseline for the experiments with six and seven input elements as provided throughout the next sections.

12.2.2. Six Input Elements

After determining a suitable weight combination for models with five input elements, a preferable weight for the sixth input element "timer array expired" is sought. The approach is similar: First, a promising weight combination considering all six input elements is determined, resulting in $IE6_{CENTER}$ (for SCO="consider" and SCO="ignore"). Then the result is sustained through investigation of the respective neighborhood.

Additionally, and different to the previous experiment, the neighborhood is not only investigated for a single protocol, but for all protocols of IE6. Thus, the results gain even more generality as they are no longer just suitable for a single protocol, but for a set of protocols.

IE6-Center.

$IE6_{CENTER}$ is determined for one of the protocols using six input elements. The RBA1 protocol (section 10.3, page 103) has been selected for this purpose. Selection of any other protocol would have been possible as well. $IE6_{CENTER}$ is derived from $IE5_{CENTER}$ by applying the $IE5_{CENTER}$ weights remaining in their tendencies and the sixth weight assuming all intermediate and matching points. Thus, the weight candidates for $IE6_{CENTER}$=("timer expired", **"timer array expired"**, "spontaneous transition", "signal with parameters", "timestep", "timer ready") are:

(w_0,$\{\boldsymbol{w_0}, \boldsymbol{w_1}, \boldsymbol{w_2}, \boldsymbol{w_3}, \boldsymbol{w_4}, \boldsymbol{w_5}\}$, w_2, w_4, w_4, w_0), for SCO="consider" and

(w_1,$\{\boldsymbol{w_0}, \boldsymbol{w_{0.5}}, \boldsymbol{w_1}, \boldsymbol{w_{1.5}}, \boldsymbol{w_2}, \boldsymbol{w_3}, \boldsymbol{w_4}, \boldsymbol{w_5}\}$, w_2, w_4, w_0, w_0), for SCO="ignore".

The experimental results for the additional input element show a clear preference for setting *wInput("timer array expired")* to $w_0 = 0$ in both cases. This is not a surprising result. In section 6.3.1 (paragraph "timer array expired", page 64) it has already been discussed that expiration of timers in arrays are most likely to represent forwarding of signals from delay processes and thus do not implement a basic fault-tolerance functionality. Thus, their impact on finding fault-tolerance violations could be assumed to be marginal.

The resulting promising weight combinations are thus:

IE6$_{CENTER}$, SCO="consider" : (w_0, $\boldsymbol{w_0}$, w_2, w_4, w_4, w_0)

IE6$_{CENTER}$, SCO="ignore" : (w_1, $\boldsymbol{w_0}$, w_2, w_4, w_0, w_0)

IE6-Neighborhood.

Table 12.14 shows the results when investigating the neighborhood of the two IE6$_{CENTER}$ weight combinations. For each of the protocols in set IE6 described throughout chapter 10, the hit-ratios (in percent) within each neighborhood are displayed.

Protocol → Neighborhood ↓	**PP**	**RBA1**	**DBA1**	**SM**	**2SP**
IE6$_{CENTER}$, SCO="consider"	95.24	60	100	90	66.67
IE6$_{CENTER}$, SCO="ignore"	88.98	75	100	63.64	22.22

Table 12.14.: IE6 Neighborhood Evaluation.

In general, the results display a good hit-ratio for all of the protocols. For completeness and comparability, the IE5-results for the pendulum protocol are displayed again − they do not change for IE6. The RBA1 protocol, serving as the basis for IE6$_{CENTER}$, shows a rather low hit-ratio in the neighborhood: 60% for SCO="consider" and 75% for SCO="ignore". However, these seemingly low ratios are rather good for this protocol as the faulty component induces a very large state space (see discussion in sections 10.3, page 103 and 12.1.3, page 131), resulting in a very low hit-ratio in general. The same holds for the 66% achieved for the 2SP protocol when Signal-Consumption-Only transitions are considered. For the SM protocol, the ratio is 90%. The very small DBA1 protocol shows 100% hits in the neighborhood. When ignoring SCOs, the results are visibly worse (except for the RBA1 protocol). For the 2SP protocol, the hit-ratio in the neighborhood is even down to 22%. This may have two different reasons: either the center-point has been chosen badly, or a clear votum for considering SCOs is, finally, visible. Additional experiments have been run to check the first possibility. They are discussed in the next paragraph.

Additional, Clarifying Experiments. If the center-point was chosen badly, selection of other, "nearly optimal" center-points should result in a better performance. For this purpose, the next three IE5 center-point candidates (ruled out in the previous section) have been taken as basis for extention to new IE6 center-points. The approach is the same as for the "real optimum", however, verification by investigation of the IE5-neighborhood has been skipped. Three additional IE6, center-points have been derived by this approach. Investigation of their neighborhoods resulted in even worse hit-ratios down to 9%. Thus it can be concluded that $IE6_{CENTER}$ has been indeed an "optimal" combination and the bad hit-ratios depicted in table 12.14 are due to the selection of SCO="ignore". Nevertheless, SCO="ignore" will be further considered in the IE7 experiments. These experiments will then substantiate or disprove this assumption.

12.2.3. Seven Input Elements

The last experiment in the series of determining suitable weights for transition input elements is extending the weight combination to all seven input elements. For this step, experiments with the FlexRay model are used as this is one of the two protocols using the input element "signals without parameters".

IE7-Center.

A suitable overall weight combination is derived from $IE6_{CENTER}$ by adding, again, the missing dimension. This results in the following candidates for $IE7_{CENTER}$=("timer expired", "timer array expired", "spontaneous transition", "signal with parameters", **"signal without parameters"**, "timestep", "timer ready"):

(w_0, w_0, w_2, w_4,$\boldsymbol{\{w_0, w_1, w_2, w_3, w_4, w_5\}}$, w_4, w_0), for SCO="consider" and

(w_1, w_0, w_2, w_4,$\boldsymbol{\{w_0, w_{0.5}, w_1, w_{1.5}, w_2, w_3, w_4, w_5\}}$, w_0, w_0), for SCO="ignore".

The results of this series of experiments show a clear preference for *wInput("signal without parameters")* $= 0$ independent of the SCO parameter. This complies with the assumption made in the "signal without parameters" paragraph of section 6.3.1 (page 64). Signals without parameters are mainly used for model internal communication and are thus not of high importance with respect to finding violations in the fault-tolerance mechanisms.

The resulting center-points for IE7 are:

$IE7_{CENTER}$, SCO="consider" : (w_0, w_0, w_2, w_4, w_0 , w_4, w_0)

$IE7_{CENTER}$, SCO="ignore" : (w_1, w_0, w_2, w_4, w_0, w_0, w_0)

IE7-Neighborhood.

Substantiating the weight combination for all seven parameters is again done through investigation of the $IE7_{CENTER}$ weight-combinations' neighborhoods. Apart from the FlexRay model, the

VETO model comprises all seven parameters. Table 12.15 displays the hit-ratios for the respective center-points. In order to give a complete overview over the performance of the complete input weight combination, the results for the protocols with less input elements are included as well. They remain the same as in the previous experiments since they do not consider the extra input element weight.

Protocol → Neighborhood ↓	**PP**	**FX**	**RBA1**	**DBA1**	**SM**	**2SP**	**VETO**
IE7$_{CENTER}$, SCO="consider"	95.24	83.33	60	100	90	66.67	92.66
IE7$_{CENTER}$, SCO="ignore"	88.98	57.45	75	100	63.64	22.22	74.89

Table 12.15.: IE7 Neighborhood Evaluation.

The hit-ratio in the promising weight-combination neighborhood of the models using all seven input elements is again different when considering SCOs and ignoring them. While the hit-ratio for the FlexRay protocol is 83.88% when SCOs are considered, it drops to 57.45% when ignoring those transitions. The same tendency is visible for the VETO protocol. A 92.66% hit-ratio is achieved for SCO="consider" while SCO="ignore" results in 74.89%.

Since the results for all of the protocols − with the exception of RBA1 − show a clear preference in favor of considering Signal-Consumption-Only transitions, the last parameter SCO="consider" is fixed.

12.2.4. Summary

Throughout section 12.2, a suitable weight combination for the seven different input elements as described in section 6.3.1 has been determined. Starting with an optimal combination considering five elements only and verifying its plausibility, suitable weights for the remaining two elements have been determined incrementally. By investigation of the neighborhood of the selected weight combinations, the suitability of this approach has been verified. It has been shown that a fix assignment of weights to the input elements increases the chances of finding fault-tolerance violations substantially for a representative set of fault-tolerant protocols. Thus it can be recommended for validation of models of fault-tolerant protocols in general.

Furthermore, a clear advantage of considering Signal-Consumption-Only transitions has become evident. Thus, the SCO parameter could be fixed to "consider". The resulting weight combination for the input weights is thus:

IE7$_{CENTER}$, SCO="consider" : $(w_0, w_0, w_2, w_4, w_0\ , w_4, w_0)$.

The next section (12.3) focuses on the action element weights (see section 6.3.2) and different combinations of action weights and input weights.

12.3. Action Weights

This section comprises all experiments concerning weights of action elements. In contrast to determining a suitable weight combination for the input elements, finding preferable weights

for action elements is more complicated as there may be more than one action element per transition. This has been discussed in detail throughout section 6.3.2. Three different functions of combining the static weights for the five action elements of each transition have been proposed:

- $f1_{action}(tr_j) = \max_{k \in \mathcal{A}_{UNIQUE}} \{wAction(action(tr_{j,k}))\}$.
  The highest static weight of the action elements occurring in tr_j is selected.
- $f2_{action}(tr_j) = \sum_{k \in \mathcal{A}_{UNIQUE}} wAction(action(tr_{j,k}))$,
  where set $\mathcal{A}_{UNIQUE} \subset \mathcal{A}$ contains only one instance of each of the action elements occurring in tr_j. Thus, $|\mathcal{A}_{UNIQUE}| \leq 5$.
- $f3_{action}(tr_j) = \sum_{k \in \mathcal{A}} wAction(action(tr_{j,k}))$.
  The weights of all k action elements constituting transition tr_j are added up.

For all experiments each of the three functions is applied, and its suitability is evaluated.

Furthermore, different weight factors for combining input and action weights of each transition are evaluated. The overall transition weight function (equation 6.11, page 70) has been defined as:

$$wTrans(tr_j) = \alpha \cdot wInput(input(tr_j)) + \beta \cdot wAction(tr_j).$$

α and β can be set such that:

- Only input weights are considered (section 12.2);
- Only action weights are considered (section 12.3.1);
- Input weights serve as add-on to the action weights (section 12.3.2);
- Action weights serve as add-on to the input weight (section 12.3.3);
- Action weights are averaged and in the same order of magnitude as the input weight (section 12.3.4).
- Single action weight and input weight (section 12.3.5).

Each of the last five input and action weight combinations is described and evaluated in detail in the indicated sections.

As for the evaluation of the input weights, all experiments have been conducted with $nDec = 0.1$, $width = 100$, $df(depth(s_i)) = 0$, $SCO =$ *"consider"*.

The approach is similar to the one for the input elements. For each setup, first a promising weight combination is determined. Then, the plausibility of the combination is substantiated by investigation of the respective neighborhood.

12.3.1. Pure Action Weights

This series of experiments aims at determining a suitable weight combination for the action elements if the weights of the input elements are ignored. Thus, α within the overall transition weight calculation function is set to zero (and $\beta = 1$), resulting in function 12.1 for this series of experiments.

$$wTrans(tr_j) = fi_{action}(tr_j), i \in \{1, 2, 3\}. \tag{12.1}$$

For each of the three functions, a suitable weight combination is determined by investigation of the model of the SM protocol (section 10.2). This weight combination is referred to as AE_P$_{CENTER}$ = ("send signal", "timer reset", "timer set", "timer array set", "variable change"), further on. The SM protocol has been selected for determining the optimal weight as it is a relatively small protocol and thus yields results rather fast. Any other protocol could have been chosen as well. In order to find AE_P$_{CENTER}$, all weight combinations are investigated (as for the input elements).

The neighborhood of AE_P$_{CENTER}$ is checked for each of the modeled protocols to ensure plausibility.

Function	Weight Combination
AE_P$_{CENTER_f1}$	(aw_2, aw_3, aw_2, aw_0, aw_1)
AE_P$_{CENTER_f2}$	(aw_2, aw_3, aw_2, aw_0, aw_1)
AE_P$_{CENTER_f3}$	(aw_2, aw_3, aw_2, aw_0, aw_1)

Table 12.16.: AE_P$_{CENTER}$ Weight Combinations.

Table 12.16 depicts the optimum weight combinations for each of the three functions $f1$, $f2$, $f3$. In order to distinguish static weights for the action elements and for the input elements, action element weights will be denoted by aw_x and input element weights by iw_y further on. The experiments for each of the three functions revealed the same (seemingly) optimal weight combination. The equality of the optimum weights does not mean selection of the function is irrelevant. When investigating the neighborhoods applying the respective functions the hit-ratios can be compared and thus the suitability of each function can be assessed.

The resulting "optimal" weights for the different action elements support the assumptions with respect to each action element weight made in section 6.3.2:

send signal: The experiments suggest to set the weight for the action element *sendSignal* to aw_2. In other words, a medium weight should be assigned to this element. Sending signals contributes to dissemination of (possibly faulty) information between nodes. Furthermore, each signal that is sent enables more transitions to be chosen in the next round. A medium weight is also preferable as sending signals results in receive operations in the next round. Thus, some weight is already given to the combined send-receive event by the input weight for signal reception. If the weight for sendSig is located in a medium range, transitions sending signals are not neglected and the input weight for signal reception can reflect the importance of the signal in combination with the consuming transitions action weights.

timer reset: The weight for timer reset actions should be the highest as this action usually expresses that a task has been completed on time.

timer set: Setting timers should be assigned a medium weight. They may either result in expiration or in resetting of the timer. The first one is reflected by the input elements *timer ready* and *timer expired*, the latter one by the action element *timer reset*. While the respective input elements are assigned a small weight (discussed throughout section 12.2.1), the *timer reset* element is assigned a high weight (previous paragraph). Thus a medium weight is preferable.

timer array set: As for all timer-array-related elements so far, the suggested weight is again zero. As timer arrays mainly occur in delay processes, their support for checking fault-tolerance abilities is very low.

variable change: *varChange* should be set to a small value. There are several kinds of variables and their setting possibilities:

1. Through parameters: this is not a "task" element of the SDL language and thus is not considered for the varChange weight. Variables set through parameters are most likely the most important ones as they may assume wrong values from sending nodes.
2. Counters are another class of variables that can be set. Counters may be a means of the fault-tolerance mechanism, e.g. counting missed tries etc. However, it is more common to use timers rather then counters. Even if counters are used, they are often manipulated upon a timeout, thus they are likely to be covered by timer events anyway.
3. Variables that are set and then distributed to other nodes. This will be captured by sendSig at the sender side and (later) by the signal reception at the other nodes. This again would result in the assumption that the varChange weight should have at most little influence on the transition weight.
4. Model internal variables: Those are most likely not to be considered relevant for the fault-tolerance mechanism as modeling faults are not subject to the investigation.
5. Other local variables: Variables that are not shared with other nodes and are not counters provide only a small impact to finding fault-tolerance violations.

Thus the influence of the action of changing variable values is not assumed to be very high. Instead, this is mainly covered by the other transition elements.

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
AE_P$_{CENTER_f1}$	66.26%	7.13%	11.75%	100%	8.33%	13.7%	9.07%
AE_P$_{CENTER_f2}$	98.72%	23.26%	55%	100%	75.5%	66.26%	15.34%
AE_P$_{CENTER_f3}$	100%	62.5%	62.5%	100%	100%	78.34%	34.21%

Table 12.17.: Neighborhood of AE_P$_{CENTER}$.

Table 12.17 summarizes the hit-ratios in the neighborhood of each AE_P$_{CENTER}$ for the modeled protocols applying the three different functions. The tendencies are clearly visible. For all

protocols, the highest hit-ratio has been achieved with the third function, while the first function performs worst. Thus, the high dissemination of resulting overall action weights achieved through AE_P$_{CENTER_f3}$ is preferable.

Due to the very clear results in favor of function three, only this function will be applied in further experiments considering input elements as well.

12.3.2. Input Weights Extend Action Weights

In the experiments presented in the previous section, no input element weights were considered. The series of experiments within this section investigates the effect of input weights extending action weights. Basically, the setup is as for the previous experiments. However, instead of assuming a weight of zero for all input elements, the optimum combination as derived in section 12.2 is selected. α and β, determining the ratio between input and action element weights are set such that input weights will only have an influence on the overall weight if the action element weights are equal. For example, assume an overall action weight of 7 and an input weight of 5. Then, the overall transition weight could be set to 7.5.

Since the optimum weight combination for the action weights has been determined in the previous section, the assumed optimum for this series of experiments can be considered the combination of these optimum weights with the optimum input weights. This also holds for the subsequent series of experiments, which will only differ in the selection of α and β.

AI$_{CENTER}$ **:** (aw_2, aw_3, aw_2, aw_0, aw_1, iw_0, iw_0, iw_2, iw_4, iw_0, iw_4, iw_0)

represents the combinations of the optimum action weights and the optimum input weights of AE_P$_{CENTER_f3}$ and IE7$_{CENTER}$.

Where the order of elements in AI$_{CENTER}$ is ("send signal", "timer reset", "timer set", "timer array set", "variable change", "timer expired", "timer array expired", "spontaneous transition", "signal with parameters", "signal without parameters", "timestep", "timer ready").

Table 12.18 summarizes the results when investigating the neighborhood of AI$_{CENTER}$.

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
AI$_{CENTER}$	100%	88.4%	66%	100%	100%	87.34%	79.5%

Table 12.18.: A$_{MAIN}$ I$_{ADDON}$ Neighborhood Results.

In all cases a clear improvement in the hit-ratios is observed − or remains if it was already 100%. The highest gain is visible for the 2SP protocol where the hit-ratio improved from 34.21% (table 12.17) to 79.5%. The overall performance increase leads to the conclusion that combining action and input weights by using the latter ones as add-on is advantageous.

12.3.3. Action Weights Extend Input Weights

The experiments of this section investigate the contrary case to the ones in section 12.3.2. Here, the input weights are preferred, while the action weights are used only to give an emphasis if two enabled transitions yield the same input weights. The center-point of the experiments is again

$\mathbf{IA}_{CENTER}$ **:** $(aw_2, aw_3, aw_2, aw_0, aw_1, iw_0, iw_0, iw_2, iw_4, iw_0, iw_4, iw_0)$;

the combination of the optimum action weight values and the optimum input weight values. The only difference to the previous experiment is selection of α and β. This time, the action weights provide the part behind the comma.

Table 12.19 provides the hit-ratios of the different protocols around IA_{CENTER}.

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
IA_{CENTER}	97.5%	87.24%	78.7%	100%	93.25%	92.66%	84.07%

Table 12.19.: I_{MAIN} A_{ADDON} Neighborhood Results.

Extending the input weights with action weights also yields a clear improvement of performance with respect to considering input weights only. The hit-ratios when considering input weights only (see table 12.15) range from 60% (RBA1) to 100% (DBA1) with an average of 83.99%. Adding action weights yields hit-ratios in the range from 78.7% (RBA1) to 100% (DBA1). The average is 90.49%.

Comparing the results to the previous experiment ($A_{MAIN}I_{ADDON}$, section 12.3.2), the average hit-ratio of the current experiment is about 2 percent points higher (increase from 88.75% to 90.49%). Thus, no significant performance improvement is visible.

12.3.4. Action Weights and Input Weights Equally

So far, it has been investigated whether extending action weights with input weights and vice versa yields performance improvements. This has been shown in the two preceeding sections. In other words, α and β in the overall weight formula have been set such that either the action weights or the input weights only provide the part behind the comma.

For the experiments in this section, α and β are chosen such that the part of the equation representing the overall action weight is in the same order of magnitude as the input weight. In other words, β is set to one and α is selected to normalize the overall action weight to "fit" the input weight. Thus, action weights and input weight have the same impact on the overall transition weight.

Again, the combination of the two optima is used as center-point

$\mathbf{EQ}_{CENTER}$ **:** $(aw_2, aw_3, aw_2, aw_0, aw_1, iw_0, iw_0, iw_2, iw_4, iw_0, iw_4, iw_0)$;

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
IA_{CENTER}	73.25%	45.78%	49.7%	100%	71%	67.24%	51.26%

Table 12.20.: EQ Neighborhood Results.

and the plausibility of this weight combination is checked by investigation of the neighborhood. Table 12.20 displays the hit-ratios of the different protocols.

The hit-ratios are clearly worse than in all of the previous experiments. For the FlexRay protocol and the RBA1 protocol they are even below 50%. In the average, a hit-ratio of 65.46% is achieved. Thus, this combination of input and action weights is considered not suitable.

12.3.5. Single Action Weights and Single Input Weights

This section discusses the last experiment concerning combinations of action and input weights. Here, the static action weights and input weights are selected in the same order of magnitude. They are merely summed up. Basically, this is close related to function

$$f\mathcal{3}_{action}(tr_j) = \sum_{k \in \mathcal{A}} wAction(action(tr_{j,k}))$$

(page 141) for the action weights. Additionally, the input weight is included in the sum. Thus, α and β are both set to one.

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
SUM_{CENTER}	77.04%	63.34%	61.21%	100%	77.04%	76.5%	52.5%

Table 12.21.: A and I Sum Neighborhood Results.

Table 12.21 shows the results around the center-point

SUM_{CENTER} : $(aw_2, aw_3, aw_2, aw_0, aw_1, iw_0, iw_0, iw_2, iw_4, iw_0, iw_4, iw_0)$.

The performance is slightly better than in the previous experiment. However, the average hit-ratio of 72.52% is still lower than those of all other experiments considering input weights and/or action weights. Thus, this weight combination cannot be suggested.

12.3.6. Summary of Weight Combinations

Throughout section 12.3, different action weight combinations have been investigated. Three functions on how to combine the weights of the action elements contained in each transition have been proposed. A clear tendency towards

$$f\mathcal{3}_{action}(tr_j) = \sum_{k \in \mathcal{A}} wAction(action(tr_{j,k})) - \text{ adding up all action weights } -$$

has been observed.

Apart from considering action weights alone, four combinations of action and input weights have been proposed. Table 12.22 summarizes the results for the protocols. The results when considering input weights only (first row) are included for completeness.

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP	Average
INPUT	95.24%	83.33%	60%	100%	90%	92.66%	66.67%	83.99%
PD	100%	62.5%	62.5%	100%	100%	78.34%	34.21%	76.79%
AI	100%	88.4%	66%	100%	100%	87.34%	79.5%	88.75%
IA	97.5%	87.24%	78.7%	100%	93.25%	92.66%	84.07%	90.49%
EQ	73.25%	45.78%	49.7%	100%	71%	67.24%	51.26%	65.46%
SUM	77.04%	63.34%	61.21%	100%	77.04%	76.5%	52.5%	72.52%

Table 12.22.: Summary of Action Weight Experiments.

The best results have been achieved when combining action and input weights such that one of the two weights is the main component of the overall weight and the other one provides the part behind the comma. Both, the AI and IA, experiments yield an average hit-ratio of about 90%. Extending the input weights with the action weights performs slightly better.

12.4. Special Transitions

Throughout this section different weights for the four special transitions `Incoming Signal in Delay Process`, `Forwarding Signal from Delay Process`, `Signal Arrival On Time`, `Sending Signal On Time` as presented in section 6.3.3 (page 72) are discussed.

In section 12.3, the best results have been achieved for the action weight function $f3$: $f3_{action}(tr_j) = \sum_{k \in \mathcal{A}} wAction(action(tr_{j,k}))$, where the weights of all k action elements constituting transition tr_j are added up. Thus, the overall action weights for the respective special transitions are:

Incoming Signals in Delay Process: $aw_{ISDP} = aw_0 + aw_1$;

Forwarding Signals from Delay Process: $aw_{FSDP} = aw_2$;

Signal Arrival on Time: $aw_{SAOT} = aw_3$;

Sending Signal on Time: $aw_{SSOT} = aw_3 + aw_2$.

The overall transition weight is derived from the best performing combination IA (section 12.3.3), where the action weight extends the input weight − denoted by iw, aw:

$wTrans(ISDP)$: iw_4 , aw_{ISDP};

$wTrans(FSDP)$: iw_0 , aw_{FSDP};

$wTrans(SAOT)$: iw_4 , aw_{SAOT};

wTrans(*SSOT*): iw_2 , aw_{SSOT}.

Assigning those weights to the special transitions yields the results presented throughout section 12.3.3.

Apart from these calculated weights, other weight assignments to the special transitions are conceivable. As discussed in section 6.3.3, `Incoming Signals in Delay Process` and `Forwarding Signals from Delay Process` should be assigned a small weight as they can be supposed to be part of the physical layer and not the fault-tolerant protocol itself. This assumption has been substantiated in the experiments so far. `Signal Arrival on Time` and `Sending Signal on Time` on the other hand should be assigned a high weight. Thus, experiments setting *wSpecial(ISDP)* and *wSpecial(FSDP)* to small values and *wSpecial(SAOT)* and *wSpecial(SSOT)* to high ones are investigated throughout the remainder of this section.

Experimental Setup. For this experiment, weights are set as follows:

$wSpecial(ISDP) = 0$;

$wSpecial(FSDP) = 0$;

$wSpecial(SAOT) > w_{highest}$;

$wSpecial(SSOT) > w_{highest}$.

$w_{highest}$ represents the highest transition weight of all transitions of the respective model.

The experimental setup follows those of the previous sections. The hit-ratios in the neighborhood are displayed in table 12.23. For better comparability, the results of the IA-experiments are

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
Hit-ratio	96.26%	89%	75.5%	100%	93.25%	94.34	83.1%
Hit-ratio (IA)	97.5%	87.24%	78.7%	100%	93.25%	92.66	84.07%

Table 12.23.: Special Transition Neighborhood Results.

provided in the last row again. The performance differences are only marginal and show no clear tendency of whether considered special transitions is favorable or not. Thus it can be suggested to refrain from differentiating between special transitions and "normal" ones.

12.5. Summary

The experiments presented throughout chapter 12 show a clear advantage of exploiting typical properties of fault tolerance techniques as applied in the new methods of the *H-RAFT* algorithm. Width restriction has been shown to perform best, when chosen small and no depth function *df* is applied (section 12.1). The experiments have substantiated the benefits of restricting the number of firing spontaneous transitions in faulty processes. While elimination

of Signal-Consumption-Only transitions has not exhibited advantages. The experiments show that combining independent optimal input and action weights drastically increases the chances of finding a fault-tolerance-violating path. The results have proven robust. Considering special transitions additionally, yields no performance improvements. This substantiates the generality of the *H-RAFT* algorithm.

H-RAFT − with the derived parameter assignments − also clearly outperforms existing algorithms. A comparison to those algorithms is provided in section 14.

13. Analysis of the Close-to-Failure Algorithm

Throughout this chapter the performance of the Close-to-Failure algorithm is evaluated. Furthermore, experiments related to the combination of C2F and H-RAFT are included and discussed. The evaluation may only cover a few exemplary definitions of the e_i and their respective relevances as the number of conceivable e_i is too large to be evaluated completely. For H-RAFT, a parameter combination suitable for *all* protocols has been sought. Since C2F is model dependent, properties have to be specified *protocol-wise*. Three experimental setups per protocol are investigated:

A1: Little user knowledge about the model is assumed.

A2: Average user knowledge about the model is assumed.

A3: Detailed user knowledge about the model is assumed.

For the first assumption **A1**, the C2F$_{PART-F}$ variant of the Close-To-Failure algorithm (see section 7.2.1) is applied. Both weight calculations for the global states are considered. Selection by e$_{MAX}$ is investigated in section 13.2, while section 13.3 provides the results when applying e$_{AVG}$ (see section 7.1, page 76). Section 13.4 discusses the experiments when combining H-RAFT and C2F weight calculation. Comparison of the Close-to-Failure algorithm to existing ones is provided in chapter 14.

13.1. Experimental Setup

This section contains the parameters and setups that are equal for all experiments and the protocol specific e_i-rules. Independent of the weight calculation functions, the **A1** to **A3** have to be defined for each protocol. Furthermore, a run-time limit of 48 CPU-hours as well as a main memory limit of 1GB are applied again to maintain comparability with the H-RAFT algorithm. It is refrained from providing all rules of the respective sets $\mathcal{E}_{A1}$, $\mathcal{E}_{A2}$, $\mathcal{E}_{A3}$ explicitly as they are only expressive if the corresponding SDL model is at hand. Instead general selection strategies for the e$_i$ constituting each set are provided and a (sometimes informal) overview of the rules is given.

The set $\mathcal{E}_{A1}$ of each of the protocols is built up through the Close-to-Failure variant C2F$_{PART-F}$ (see section 7.2.1, page 78). Table 13.1 provides the number of resulting rules in $\mathcal{E}_{A1}$ for each protocol. The weights for each of the e_i is not specified here, but in the setups of the respective experiments (tables 13.2 to 13.8). The rules of the second set $\mathcal{E}_{A2}$ should represent an average model knowledge. Such rules may be specified, for example, by a user who knows the protocol, but has not seen the model code so far. Straightforward selection strategies may comprise, for example, local states containing the words “faulty” or “error” etc. Also rules of $\mathcal{E}_{A1}$ are often

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
$\lvert\mathcal{E}_{\mathbf{A1}}\rvert$	3	2	2	2	6	2	4
$\lvert\mathcal{E}_{\mathbf{A2}}\rvert$	8	11	6	3	10	5	13
$\lvert\mathcal{E}_{\mathbf{A3}}\rvert$	13	23	9	6	14	8	17

Table 13.1.: Number of Rules in $\mathcal{E}_{A1}$.

found in $\mathcal{E}_{A2}$, sometimes combined with additional knowledge and so on. Usually, no complex interactions or execution paths are included in the e_i of $\mathcal{E}_{A2}$ as this often requires in-depth model knowledge. Such rules are assumed in $\mathcal{E}_{A3}$ along with rules specified in $\mathcal{E}_{A2}$ already. Thus, the number of rules in $\mathcal{E}_{A3}$ is usually (but not necessarily) larger than the one in $\mathcal{E}_{A2}$. The numbers of specified rules in $\mathcal{E}_{A2}$ and $\mathcal{E}_{A3}$ for each protocol are also shown in table 13.1.

In the following tables, an informal overview over the rules in the three sets is provided for each protocol. The tables depict the conditions of the rules in the first column. The other columns indicate the assigned weights if the respective assumption is applied. Empty cells represent a weight of zero indicating the rule is not used for the assumption.

Conditions of the rule	**A1**	**A2**	**A3**
The values of the fault-free nodes differ at some time	0.7	0.5	0.12
The evaluation process is in state reception_error	0.1	0.03	0.005
The evaluation process is in state red	0.2	0.12	0.01
Node A is in state i_m_faulty		0.05	0.02
Node A is in state i_m_faulty_send_to_B		0.1	0.04
Node A is in state i_m_faulty_send_to_C		0.1	0.04
Node B is in state i_m_faulty		0.05	0.02
Node C is in state i_m_faulty		0.05	0.02
Comm_timer in node A expires			0.08
Comm_timer in node B expires			0.185
Comm_timer in node C expires			0.25
ValueC is received at node B in state wait_for_value_A_or_T			0.1
ValueA is received at node B in state wait_for_value_C			0.1

Table 13.2.: Rules of the Pendulum Protocol.

Conditions of the rule	**A1**	**A2**	**A3**
The values of the fault-free nodes differ at some time	0.3	0.2	0.1
Max_timer expires at evaluation process	0.7	0.3	0.05
Value “unknown” is received at a node		0.5	0.15
Variable “meist” is altered			0.25
It is an odd round			0.35
The final value is decided on by a node			0.1

Table 13.3.: Rules of the DBA1 Protocol.

Conditions of the rule	**A1**	**A2**	**A3**
The system is in state yellow	0.75	0.07	0.005
The system is in state red	0.25	0.05	0.005
A node is in state header_failed		0.12	0.04
A node is in state header_failed_unknown		0.11	0.03
A node is in state header_failed_unknown_idle_possible		0.09	0.015
A node is in state fail_silent		0.2	0.06
A node is in state fatal_state		0.1	0.03
A node is in state FR_abort		0.09	0.02
A node is in state suspend		0.1	0.03
Error management receives unexpected syncframe		0.04	0.01
Signal panic is received		0.03	0.005
Variable mocs in the error management is altered			0.01
Variable mrcs in the error management is altered			0.01
The system is in the second cycle			0.05
A node is in state header_end_simult_unknown			0.04
A node is in state sending_idle_unknown			0.04
A node is in state sending_tss_unknown			0.04
A node is in state wait_for_idle_unknown			0.04
A node is in state wait_for_unknown_idle_possible			0.04
A transmission occurs during the NIT			0.15
An invalid TSS is received			0.12
An invalid header is received			0.11
An invalid payload is received			0.1

Table 13.4.: Rules of the FlexRay Protocol.

Conditions of the rule	**A1**	**A2**	**A3**
The values of the fault-free nodes differ at some time	0.3	0.18	0.08
Max_timer expires at evaluation process	0.7	0.2	0.02
A node is in state is_faulty		0.1	0.05
A node is in state design_fault		0.07	0.03
A node is in state design_fault_wait		0.05	0.02
Value “unknown” is received at a node		0.4	0.1
Variable “meist” is altered			0.25
It is an odd round			0.35
The final value is decided on by a node			0.1

Table 13.5.: Rules of the RBA1 Protocol.

Conditions of the rule	**A1**	**A2**	**A3**
Values in the consistency vector are modified	0.35	0.17	0.01
A node decides on its final value	0.25	0.05	0.01
Node A has reached its final state	0.1	0.02	0.01
Node B has reached its final state	0.1	0.02	0.01
Node C has reached its final state	0.1	0.02	0.01
Node D has reached its final state	0.1	0.02	0.01
The first signature is invalid		0.25	0.16
The second signature is invalid		0.2	0.13
The third signature is invalid		0.15	0.11
The fourth signature is invalid		0.1	0.07
Two signatures are equal			0.10
Three signatures are equal			0.12
Four signatures are equal			0.15
The consistency vector contains an invalid value			0.1

Table 13.6.: Rules of the Signed Messages Protocol.

Conditions of the rule	**A1**	**A2**	**A3**
Variable isOk in AK is altered	0.7	0.1	0.05
Variable r1Val is altered	0.1	0.03	0.007
Variable r2Val is altered	0.1	0.03	0.007
Variable r3Val is altered	0.1	0.03	0.007
R1 is in state is_faulty		0.05	0.005
S1 is in state is_faulty		0.05	0.005
S1 is in state is_faulty_wait		0.03	0.002
R2 is in state is_faulty		0.05	0.005
R3 is in state is_faulty		0.05	0.005
S2 is in state is_faulty		0.05	0.005
S2 is in state is_faulty_wait		0.03	0.002
Variable compareOK is set to false in R2		0.3	0.15
Variable sendAgain is set to true in R2		0.2	0.11
R1 forwards its value at the latest allowed time			0.2
R2 finishes comparison at the latest allowed time			0.17
R2 is in state wait_again			0.1
R3 finishes comparison at the latest allowed time			0.17

Table 13.7.: Rules of the 2-Switch Protocol.

Conditions of the rule	**A1**	**A2**	**A3**
Variable wait_for_faulty in evaluation process is modified	0.4	0.1	0.02
Max_duration_timer in evaluation process expires	0.6	0.15	0.01
Node in state wait_for_veto		0.25	0.05
Node in state model_error		0.1	0.02
Node in state decide_on_veto		0.5	0.1
Message arrives during state wait_for_veto			0.2
Veto received by a node			0.28
Value deviation detected			0.32

Table 13.8.: Rules of the VETO Protocol.

13.2. Global State Selection by e_{MAX}

Within this section, experiments are discussed where the global state weight calculation is done by applying e_{MAX}. e_{MAX} represents the weight of the e_i in $\mathcal{E}$ that has the highest assigned weight of all fulfilled e_i (see section 7.1). The results for all protocols are shown in table 13.9. For each protocol − when applying the rules of $\mathcal{E}_{A1}$, $\mathcal{E}_{A2}$ and $\mathcal{E}_{A3}$ respectively − it is indicated whether a fault-tolerance violation has been found ("FOUND") or has been missed ("−").

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
$\mathcal{E}_{\mathbf{A1}}$	−	−	−	FOUND	−	−	−
$\mathcal{E}_{\mathbf{A2}}$	FOUND	−	−	FOUND	−	−	−
$\mathcal{E}_{\mathbf{A3}}$	FOUND	−	−	FOUND	−	−	FOUND

Table 13.9.: Results for e_{MAX}.

When applying **A1**, the fault-tolerance violation has been found for the very small DBA1 protocol only. The additional rules defined for **A2** result in a violation being found for the pendulum protocol as well. A violation in the 2-Switch protocol is discovered when the rules of **A3** are considered.

In summary: applying e_{MAX} for the global state weight selection yields very low chances of finding fault-tolerance violations.

13.3. Global State Selection by e_{AVG}

This section presents the experiments and their discussion when e_{AVG} is used for the global state weight calculation. e_{AVG} represents the sum of all weights of the $e_i \in \mathcal{E}$ that are currently fulfilled, divided by the number of fulfilled rules (see section 7.1). Table 13.10 provides the results of the experiments when applying the rules of $\mathcal{E}_{A1}$, $\mathcal{E}_{A2}$ and $\mathcal{E}_{A3}$ to each of the protocols, respectively.

As for e_{MAX}, a violation has only been found for the DBA1 protocol when applying the rules of **A1**. **A2** yields two additional hits: for the pendulum protocol and the 2-Switch protocol.

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
$\mathcal{E}_{\mathbf{A1}}$	–	–	–	FOUND	–	–	–
$\mathcal{E}_{\mathbf{A2}}$	FOUND	–	–	FOUND	–	–	FOUND
$\mathcal{E}_{\mathbf{A3}}$	FOUND	–	–	FOUND	–	FOUND	FOUND

Table 13.10.: Results for e_{AVG}.

The latter hit has not been observed when e_{MAX} has been applied in conjunction with **A2**. Considering the rules of **A3**, a violation in the VETO protocol has been detected additionally.

Applying e_{AVG} for the global state weight selection outperforms e_{MAX}. However, the results are still not convincing.

13.4. Combination of H-RAFT and C2F

The performance when applying Close-to-Failure stand-alone is rather poor. Thus, a combination of the H-RAFT and the C2F algorithm is introduced. Experiments are conducted and the results are discussed. The two-step selection strategy for selecting the next transition to be executed is applied again. The following paragraphs describe the applied combinations for calculating the overall transition weights and the overall global state weights.

Transitions Weight Composition. A combination of the transition weight calculations of the two algorithms has been proposed in section 7.3: The Close-to-Failure weight is selected if it has been defined. Otherwise, the H-RAFT weight is applied. H-RAFT transition weights are calculated by extending the action weights by input weights as this combination yields the best results (see section 12.3.3). H-RAFT action weights are calculated according to function $f3_{action}$ (see section 12.3, page 141). The rules applied with respect to the Close-to-Failure part of the combination are those of **A3** of the respective protocol.

Global State Weight Composition. Calculating the global state weight through applying e_{AVG} has proven advantageous for the Close-to-Failure algorithm. In H-RAFT the global state weight is based on the transition weight. Straightforward combinations of the H-RAFT and C2F global state weight functions are:

HC$_{\mathsf{SUM}}$: The sum of the two weights is selected;

HC$_{\mathsf{C2F}}$: The Close-to-Failure weight is selected if larger than zero, otherwise the H-RAFT weight is chosen. This corresponds to the approach applied for calculating the overall transition weight.

HC$_{\mathsf{AVG}}$: The average of the two weights is selected;

HC$_{\mathsf{MAX}}$: The highest of the two weights is selected.

For these experiments, the H-RAFT weights and the C2F weights are selected in the same order of magnitude.

Table 13.11 displays the experimental results for each protocol when applying the four global state weight combinations HC respectively. Again, it is denoted whether a fault-tolerance violation has been found or missed. The last two rows contain the results of the best-performing pure Close-to-Failure respectively pure H-RAFT variant.

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
$\mathbf{HC_{SUM}}$	FOUND	–	–	FOUND	–	–	FOUND
$\mathbf{HC_{C2F}}$	FOUND	–	–	FOUND	–	–	–
$\mathbf{HC_{AVG}}$	FOUND	–	FOUND	FOUND	–	–	FOUND
$\mathbf{HC_{MAX}}$	FOUND	–	–	FOUND	FOUND	FOUND	FOUND
$\mathcal{E}_{\mathbf{A3}}$ **(AVG)**	FOUND	–	–	FOUND	–	FOUND	FOUND
$\mathbf{IA_{CENTER}}$	97.5%	87.24%	78.7%	100%	93.25%	92.66%	84.07%

Table 13.11.: Results for Combined H-RAFT and C2F Weight Calculation.

Summarizing the results of the four HC variants, $\mathbf{HC_{MAX}}$ outperforms the other ones followed by $\mathbf{HC_{AVG}}$ and $\mathbf{HC_{SUM}}$. $\mathbf{HC_{C2F}}$ performs worst.

The results in summary show hardly any improvements with respect to the basic Close-to-Failure algorithm. When applying $\mathbf{HC_{AVG}}$ an additional violation has been found for the RBA1 protocol. Another additional hit is observed for the SM protocol when using $\mathbf{HC_{MAX}}$. Several misses are counted where the pure C2F algorithm has achieved a hit: For the 2SP protocol with $\mathbf{HC_{C2F}}$ and for the VETO protocol with all variations except $\mathbf{HC_{MAX}}$. In summary, only the combination $\mathbf{HC_{MAX}}$ provides a small improvement compared to C2F alone.

The H-RAFT row contains the hit-ratios in the neighborhood of the "optimum" weight combination since, for all protocols, a hit has been observed when applying the optimum. Thus, adding C2F weights even decreases the chances of finding fault-tolerance violations. Comparing the percentages and the results of the HC-combinations, it can be observed that − with the exception of the VETO protocol − the number of violations found in the respective column increases with higher percentages. Thus, it can be concluded that the basic factor for hits or misses is the H-RAFT weight. The C2F inclusion in the overall weight calculation only "disturbs" this weight.

14. Comparison of the Algorithms

This chapter focuses on the comparison of the novel algorithms H-RAFT and C2F (and their combination) to the wide-spread algorithms Random, Exhaustive, Bitstate. First the performance of the latter three algorithms is summarized in section 14.1. Those results are then compared to the novel algorithms in section 14.2 and a summary of the experiments is provided.

14.1. Random, Exhaustive, Bitstate Results

A single random walk creates exactly one path through the reachability graph (see section 3.2.2). Thus, a representative number of runs has to be conducted to get profound results. For the experiments, 10,000 runs have been sufficient to provide solid results. The percentage of paths where a fault-tolerance violation has been found is provided in table 14.1 for each of the protocols.

The experiments for the exhaustive algorithm (section 3.2.1) and the bitstate algorithm (section 3.2.3) are, again, limited to 1GB main memory and 48 CPU-hours run-time. Since the global state spaces generated by these two algorithms are based on a depth-first traversal, the reachability graph is (possibly) restricted in its width (see also figure 3.4, page 29). Only a single run is required for these algorithms as they are deterministic. The results of the "resource-restricted" bitstate and exhaustive algorithms are also included in table 14.1. They are binary as either a fault-tolerance violation has been found (indicated by "FOUND") or not (indicated by "–").

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
Random	0.01%	0%	0%	87.24%	27.50%	0.14%	1.43%
Bitstate	FOUND	–	–	FOUND	FOUND	–	FOUND
Exhaustive	FOUND	–	–	FOUND	–	–	–

Table 14.1.: Random, Bitstate, Exhaustive Results.

The performance of all three algorithms is rather poor. The exhaustive algorithm only found a violation for the the small models of the Pendulum Protocol and the DBA1 protocol due to its high requirements of main memory. The bitstate algorithm yields slightly better results. Apart from the two very small models, hits for the SM protocol and the 2SP protocol have been observed. The random algorithm shows only a marginal number of hits for most of the protocols. For the DBA1 protocol 87.24% of the investigated paths contained a fault-tolerance violation.

14.2. Comparison of the Algorithms

Table 14.2 contains the results of all algorithms. For completeness, the results of the random, exhaustive and bitstate algorithms are included again.

For the novel algorithms, the best-performing variant has been selected. Thus, C2F refers to the $\mathcal{E}_{\mathbf{A3}}$ version applying e_{AVG}. H-RAFT resembles the action weight function $f3_{action}$ and the overall transition weight is derived by extending the action weights with the input weights. HC represents the combination of the two novel algorithms applying HC_{MAX}.

Protocol	PP	FX	RBA1	DBA1	SM	VETO	2SP
Random	0.01%	0%	0%	87.24%	27.50%	0.14%	1.43%
Bitstate	FOUND	–	–	FOUND	FOUND	–	FOUND
Exhaustive	FOUND	–	–	FOUND	–	–	–
C2F	FOUND	–	–	FOUND	–	FOUND	FOUND
HC	FOUND	–	–	FOUND	FOUND	FOUND	FOUND
H-RAFT	97.5%	87.24%	78.7%	100%	93.25%	92.66%	84.07%

Table 14.2.: Comparison of All Algorithms.

For the FlexRay and the RBA1 protocol, only H-RAFT found a violation. These two protocols result in the largest state space: The FlexRay protocol because it is by far the most complex protocol. For the RBA1 model, the implemented faulty behavior leads to an enormous state space growth since the "any output at any time" paradigm is not restricted here.

The pure Close-to-Failure algorithm provides a performance in the range of the bitstate algorithm. While the bitstate algorithm found a fault-tolerance violation in the SM protocol, the C2F yields a hit for the VETO protocol. Compared to the random and the exhaustive algorithm, C2F results in a better performance.

Since the combined HC algorithm outperforms the C2F algorithm, it also provides better results than the random and exhaustive algorithms. Furthermore, one additional violation − for the VETO protocol − has been detected than with the bitstate algorithm.

The H-RAFT row contains, again, the percentages of hits in the neighborhood of the "optimum" weight combination. Applying the optimum weight combination yields hits for all protocols. In other words: each cell would contain "FOUND". Thus, H-RAFT outperforms all of the other protocols. The number of hits in the neighborhood surrounding the "optimum" shows, additionally, the robustness of this algorithm.

Summary and Conclusion. Both novel algorithms, as well as their combination, perform better with respect to finding fault-tolerance violations than the random and the exhaustive algorithm. While the C2F algorithm yields a performance in the range of the bitstate algorithm, the combined approach as well as the H-RAFT algorithm outperform all of the other algorithms. The H-RAFT algorithm yields by far the highest chances of finding fault-tolerance violations.

Although the Close-to-Failure algorithm *explicitely* takes the fault-tolerance properties highly into account, it performs surprisingly bad compared to the static H-RAFT algorithm. Due to

the mapping of transition elements to weights in H-RAFT, fault-tolerance properties are also taken highly into account (if the model is not too poor), although only *indirectly*.

In case the optimum weight combination of H-RAFT does not yield a hit and additional runs are desired, two strategies may be applied: either weight combinations in the neighborhood could be selected since the hit-ratios in the neighborhood are rather high, or the combined HC algorithm is applied as it may cover parts of the model not reached through H-RAFT alone.

Part V.

Summary and Future Work

15. Summary and Conclusions

When coping with models of large fault-tolerant protocols, the state space explosion problem is the most present one. Throughout this thesis strategies, techniques and algorithms have been introduced and investigated to tackle the problem of analyzing huge state spaces for models of fault-tolerant protocols. The goal behind all mechanisms is to find loopholes in the fault-tolerance mechanisms of the protocols. Common to all contributions of this thesis is that they exploit the fault-tolerant nature of the protocols.

A first step for reducing the state space is to apply a partial ordering strategy. The single-fault-region-oriented partial ordering (SFR-PO) algorithm has been presented and discussed in chapter 5.1. This approach reduces the concurrency of faults on one side and the fault tolerance algorithm on the other. The reduction is limited to the paths which are not of interest for the model analysis. A solution has been developed to apply this approach to standard SDL. A reduction of the global state space of more than 50% for all of the considered protocols has been achieved.

By some simple model structuring concepts concerning start transitions (see section 5.2), fault location processes and evaluation processes (see section 5.3) the state space is reduced further.

Although the state space is reduced through these techniques, it is still too large to be explored completely in most cases. Thus, algorithms for "smart" partial exploration are required. These algorithms provide heuristics for finding loopholes in the fault-tolerance mechanisms. Two algorithms H-RAFT and C2F for (partial) heuristic reachability analysis have been designed. Both algorithms are based on weights for transitions and global states. These weights are meant to guide the analysis to "interesting" parts. Both algorithms exploit the fault-tolerant nature of the protocols.

The H-RAFT (heuristic reachability analysis of fault-tolerant systems) algorithm is based on SDL elements constituting each transition. Thus, static transition weights can be computed off-line. Information dependent on the exploration progress like the current time are included in the global state weight calculation. Different weight combinations have been investigated and an optimum weight combination has been derived. This combination has been applied to all implemented protocols. The fault-tolerance violation has been detected for each of the protocols. Compared to standard algorithms (exhaustive, bitstate, random), the performance increase is clearly visible. While H-RAFT found the violations for each protocol, the well-known algorithms detected them only for 50% of the protocols in the average.

Although the H-RAFT algorithm provides very good results, it could be expected that including additional user-knowledge about the model / protocol yields even better results. Thus, the Close-to-Failure algorithm has been designed.

The Close-to-Failure algorithm is heavily guided by user-knowledge. The user may specify properties indicating its perception on how close to a fault the system is if the respective property

is fulfilled. For this purpose (different) weights may be assigned to each property. According to these weights the walk through the state space is guided. Although it could be expected that this explicit specification of fault-tolerance properties increases the chances of finding violations, this was only observed comparing it to the resource-limited exhaustive and random algorithms. The performance of C2F is in the range of the bitstate algorithm and clearly below the H-RAFT algorithm.

Since the C2F algorithm itself did not provide satisfactory results, it has been combined with the H-RAFT algorithm: adding (valuable) user-knowledge to an already well-performing algorithm. Combining the two novel algorithms results in a better performance than the Close-to-Failure algorithm alone. However, in summary the H-RAFT algorithm stand-alone outperforms this combination as well.

Integration of the novel techniques and algorithms into existing tools has proven nearly impossible. Thus, the RAFT tool has been developed. RAFT is a modular framework for implementing reachability analysis algorithms. All novel techniques and algorithms are provided by RAFT as well as the three standard algorithms. Interfaces for easy insertion of additional algorithms are provided. The major objective of the tool is to provide a platform for evaluation of the novel mechanisms.

The work has demonstrated that finding loopholes in the fault-tolerance mechanisms of large protocols can be visibly improved by exploiting the fault-tolerant nature of the protocols. This holds for pre-shrinking the state space through partial ordering techniques as well as for guided partial exploration strategies. Furthermore, it can be refrained from requiring user-knowledge about the protocol or the model since SDL-element-based guidance yields the highest success-rate in finding violations of the fault-tolerance properties.

16. Future Work

In this chapter some future research directions are identified. The field of subsequent works is vast, thus only a rough overview may be provided. Generally, research may be continued in three categories:

1. Development of additional algorithms and techniques;
2. Improvement of the novel algorithms and techniques;
3. Improvement of the RAFT-tool;
4. Applied Research.

Development of Additional Algorithms and Techniques. The first category is by far the widest field. Any technique shrinking the state space to the "interesting" parts fall into this category. Conditions to exclude certain parts of the state space from exploration are innumerable. This may reach from explicit definition of states to purely randomized strategies.

Adaption of existing algorithms to fault-tolerance protocols may be considered as well as extending mechanisms to evaluate protocols with a different purpose than finding fault-tolerance loopholes. For example, performance evaluation may be combined with checking the fault-tolerance mechanisms.

It is also conceivable to add algorithms for automatically generating test-cases. After the analysis suggestions may be provided for fault-injection experiments into the (hardware-) implemented protocol. Also improvements of the protocol could be suggested or automatically inserted if certain faults have been detected. Into the same vein, results from fault-injection may be used as input for reachability analysis algorithms. Thus forming a loop for testing and improving protocols semi-automatically.

Improvement of the Novel Algorithms and Techniques. Instead of developing new techniques, improvements of the novel algorithms could be sought. For the H-RAFT algorithm, an approach may be to consider additional functions for combining the different weights. More special transitions (see section 6.3.3, page 72), representing typical fault-tolerance or model-internal transitions, may be identified and assigned specific weights.

A straightforward direction for the Close-to-Failure algorithm would be to provide means for the user to specify more valuable rules. Either by pre-analyzing the static model and making suggestions about weights and typical properties or by analysis of the exploration run. In the latter case, improvements of the rules may be suggested to the user for a subsequent analysis.

For both algorithms, self-tuning mechanisms may be included. By learning from previous runs, weights may be adapted or "valuable" properties may be derived for future explorations.

Another branch of extending the algorithms is to combine them with other existing algorithms. For example, both algorithms could be combined with the bitstate algorithm. Thereby both speed and memory limitations are reduced resulting in a larger portion of the state space that can be explored. However, the bitstate algorithm may lead to unpredicted path-cuts possibly resulting in omission of "interesting" parts. No assumption can be made (without investigation) whether these cuts nullify the advantage of exploring more states.

Finally, a research direction could aim at finding an optimal combination between the H-RAFT and the C2F techniques such that a beneficial combination of user-knowledge and static weights is achieved.

Improvement of the RAFT-Tool. The last field of future research directions discussed here is to improve the RAFT tool. This goal can be, again, divided in three categories:

1. Provide more user-friendliness;
2. Provide more functionality;
3. Improve efficiency.

The purpose of the tool has been to provide a basis for comparison of the algorithms and techniques. Thus, user-friendliness has been a secondary goal. In future versions a graphical user-interface is aspired. Furthermore, means should be provided to allow for easy specification of properties and rules.

Additional features may also lead to a more convenient tool. An O/R-Mapper may be integrated to store evaluation runs and continue them at any time. An interface is already provided in RAFT. Although an MSC-Viewer is available for RAFT, the current output format is aimed at fast and script-based evaluation of the experimental results. Thus, an option should be provided to store the exploration runs as MSCs. Again, an interface is already provided in RAFT for this purpose. An additional thought may be to extend the amount of recognized SDL.

Extentions to the tool may also comprise importing and exporting models specified in other languages.

Apart from making the tool more convenient, efficiency improvements may also be pursued. For this purpose, data-structures may be reconsidered, libraries containing fast code for frequent operations could be incorporated or implemented.

Applied Research. Additionally to the research directions discussed so far, real-world applications could be considered. In the automotive sector, the suitability of the approaches for analyzing huge protocols has already been proven by investigation of the FlexRay protocol. Further application areas may be identified and the techniques may be adopted to fit the needs of the respective industrial branches. Exploiting the knowledge about different areas (like considering chemical processes etc.) it may be possible to develop more specific solutions. Also,

integrating and establishing validation as a step in (existing) product-development processes is conceivable. It may even be aimed at creating new development or design processes due to the availability of specific validation algorithms.

In summary, the contributions of this thesis already provide substantial performance improvements in the analysis of fault-tolerant systems. Nevertheless, it provides also an environment and a basis to conceive and integrate further approaches.

Bibliography

[AALC92] D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet. Fault Injection for the Formal Testing of Fault Tolerance. In *Proceedings of FTCS-22*, pages 345–354, Boston, MA, USA, 1992. IEEE Press.

[ABCS01] Laurent Arditi, Hédi Boufaïed, Arnaud Cavanié, and Vincent Stehlé. Coverage Directed Generation of System-Level Test Cases for the Validation of a DSP System. *Lecture Notes in Computer Science*, 2021:449–464, 2001.

[ABH+97] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order Reduction in Symbolic State-space Exploration. In *Proceedings of the 9th International Conference of Computer-aided Verification*, LNCS 1254, pages 340–351, 1997.

[ACG96] Joanne M. Atlee, Marsha Chechik, and John D. Gannon. Using Model Checking to Analyze Requirements and Design. *Advances in Computers*, 43:141–178, 1996.

[ACH+96] S. Ayache, E. Conquet, Ph. Humbert, C. Rodriguez, J. Sifakis, and R. Gerlich. Formal Methods for the Validation of Fault Tolerance in Autonomous Spacecraft. In *FTCS*, pages 353–357, 1996.

[AHH96] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[BBB+00] R. Belschner, J. Berwanger, C. Bracklo, Chr. Ebner, B. Hedenetz, W. Kuffner, P. Lohrmann, J. Minuth, M. Peller, A. Schedl, and V. Seefried. Requirements Towards an Advanced Communication System for Fault-Tolerant Automotive Applications. In German: Anforderungen an ein zukünftiges Bussystem für fehlertolerante Anwendungen aus Sicht der Kfz-Hersteller. In *9th International Conference on Electronic Systems for Vehicles*, Baden-Baden, Germany, October 2000. (in German).

[BBG04] T. Basten, D. Bosnacki, and M.C.W. Geilen. Cluster-Based Partial-Order Reduction. *Automated Software Engineering*, 11(4):365–402, October 2004.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer-Verlag, 1998.

[BE04] S. Böhm and K. Echtle. State Space Reduction in SDL Models of Fault-Tolerant Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 206, Santa Fe, NM USA, 2004. IEEE Press.

[BF95] I.S. Bonatti and R.J.O. Figueiredo. An Algorithm for the Translation fo SDL into Syntesizable VHDL. *Current Issues in Electronic Modeling*, 1995.

[BFG+99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *World Congress on Formal Methods (1)*, pages 307–327, 1999.

[BFG02] C. Bernardeschi, A. Fantechi, and S. Gnesi. Model Checking Fault Tolerant Systems. *Software Testing, Verification Reliability*, 12(4):251–275, 2002.

[BGK+96] J. Bengtsson, W.O.D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 244–256, New Brunswick, NJ, USA, 1996. Springer Verlag.

[BGPQ02] S. Boroday, R. Groz, A. Petrenko, and Y. Quemener. Techniques for Abstracting SDL Specifications. In *Proceedings of the Third SAM (SDL And MSC) Workshop*, pages 141–157, 2002.

[BK02] F. Bause and P. Kritzinger. *Stochastic Petri-nets.* Vieweg publishing house, 2^{nd} edition, 2002.

[BLL+96] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 431–434, 1996.

[Blo01] Roderick P. Bloem. *Search Techniques and Automata for Symbolic Model Checking.* PhD thesis, Rijksuniversiteit Leiden, the Netherlands, 2001.

[BMU98] J.A. Bergstra, C.A. Middelburg, and Y.S. Usenko. Discrete Time Process Algebra and the Semantics of SDL. Technical Report SEN-R9809, CWI, June 1998.

[Böh05] S. Böhm. H-RAFT – Heuristic Reachability Analysis for Fault Tolerance Protocols Modelled in SDL. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, pages 466–475, Yokohama, Japan, 2005.

[Bro91] Manfred Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing*, 3(1):21–57, 1991.

[BT97] D. Blough and T. Torii. Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message Passing Parallel Computers. In *FTCS-27*, pages 258–267, 1997.

[CAB+98] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon Damon Reese. Model Checking Large Software Specifications. *Software Engineering*, 24(7):498–520, 1998.

[CCO02] J.M. Cobleigh, L.A. Clarke, and L.J. Osterweil. FLAVERS: A Finite State Verification Technique for Software Systems. *IBM Systems Journal*, 41(1):140–165, 2002.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, 1999. Chapter 10.

[CGP02] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software Model Checking in Practice: An Industrial Case Study, 2002.

[CS97] Ana Cavalli and Amardeo Sarma, editors. *Proceedings of the 8th SDL Forum: SDL '97 — Time for Testing — SDL, MSC and Trends*, Evry, France, 23rd–26th September 1997. Elsevier Science Publishers.

[CT97] G. Csopaki and K. Turner. Modelling Digital Logic in SDL. In *Proceedings of the Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'97)*, pages 367–382, 1997.

[CW00] Marsha Chechik and Hai Wang. Feasibility of Bisimulation Analysis of Protocols Expressed in SDL. In *Proceedings of CASCON'2000*, pages 65–77, November 2000.

[DHMC96] M. Diefenbruch, J. Hintelmann, and B. Müller-Clostermann. The QUEST Approach for the Performance Evaluation of SDL-Systems. In R. Gotzhein and J. Bredereke, editors, *Proceedings of IFIP TC6 / 6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI*. Chapman & Hall, 1996.

[DJMT96] S. Dawson, F. Jahanian, T. Mitton, and L. Tung. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *FTCS-26*, pages 404–414, 1996.

[DM04] Peter C. Dillinger and Panagiotis Manolios. Fast *and* Accurate Bitstate Verification for SPIN. In *11th SPIN Workshop on Model Checking Software*, volume 2989 of *LNCS*. Springer-Verlag, April 2004.

[DMIJ97] J.-M. Daveau, G.F. Marchioro, T. Ben Ismail, and A.A. Jerraya. COSMOS: An SDL Based Hardware/Software Codesign Environment. *Current Issues In Electronic Modeling*, 8:59–88, 1997.

[DMVJ93] J.-M. Daveau, G.F. Marchioro, C.A. Valderrama, and A.A. Jerraya. VHDL Generation from SDL Specifications. In C. Delgado-Kloos and E.Cerny, editors, *Proceedings of Computer Hardware Description Languages and their Applications XIII*, pages 20–25, London, UK, April 1993. Chapman-Hall.

[DY95] C. Daws and S. Yovine. Two Examples of Verification of Multirate Timed Automata with KRONOS. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95),* Pisa, Italy, pages 66–75, 1995.

[Ech84] K. Echtle. *Fehlermodellierung bei Simulation und Verifikation von Fehlertoleranzalgorithmen für verteilte Systeme*, chapter 73-88. Number 83 in Informatik Fachbericht. Springer, 1984. (in German).

[Ech87] K. Echtle. Fault Masking and Sequence Agreement by a Voting Protocol with Low Message Number. In *Proceedings of the RDS-6*, pages 149–160, 1987.

[Ech89] K. Echtle. Distance Agreement Protocols. In *Proceedings of FTCS'89*, pages 191–198. IEEE Computer Society Press, 1989.

[Ech90] K. Echtle. *Fehlertoleranzverfahren*. Studienreihe Informatik. Springer Verlag, Berlin, Heidelberg, 1990. (in German).

[EL95] K. Echtle and M. Leu. Test of Fault-Tolerant Distributed Systems by Fault Injection. In *Proceedings of FTPDS'95*, pages 10–17. IEEE Press, 1995.

[ELL01] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF–SPIN. *Lecture Notes in Computer Science*, 2057:57–78, 2001.

[EM96] K. Echtle and A. Masum. A Multiple Bus Broadcast Protocol Resilient to Non-Cooperative Byzantine Faults. In *Proceedings of FTCS-26*, pages 158–167, 1996.

[EM00] K. Echtle and A. Masum. A Fundamental Failure Model for Fault-Tolerant Protocols. In *IEEE International Computer Performance and Dependability Symposium (IPDS2K)*, pages 69–78, Chicago, USA, 2000. IEEE.

[EN99] K. Echtle and T. Nikolov. Testing Safety Techniques by Injection of Systematically Selected Errors. In *ARCS '99*, pages 127–140, 1999.

[Fle02] FlexRay Consortium. FlexRay International Workshop. http://www.flexray.com, 2002.

[GF93] Mohammed Ghriga and Phyllis G. Frankl. Adaptive Testing of Non-Deterministic Communication Protocols. In *Protocol Test Systems*, pages 347–362, 1993.

[GKRM93] W. Glunz, T. Kruse, T. Rössel, and D. Monjau. Integrating SDL and VHDL for System-Level Hardware Design. In *Proceedings of Computer Hardware Description Languages XI*, pages 187–204, Amsterdram, Netherlands, April 1993. North-Holland.

[God91] J.C. Godskesen. An Operational Semantic Model for Basic SDL. Technical Report TFL RR 1991-2, Tele Danmark Research, 1991.

[God97] Patrice Godefroid. Model Checking for Programming Languages using Verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[God03] P. Godefroid. Software model checking: the VeriSoft approach, 2003.

[GRK93] W. Glunz, T. Rössel, and T. Kruse. Hardware/Software Co-Design Using SDL. In *Proceedings of the 2nd International Workshop on Hardware/Software Co-Design*, pages 5–21, Innsbruck, Austria, May 1993.

[GSTH96] J. Grabowski, R. Scheurer, D. Toggweiler, and D. Hogrefe. Dealing with the Complexity of State Space Exploration Algorithms. In *Proceedings of the 6th GI/ITG technical meeting on 'Formal Description Techniques for Distributed Systems'*, University of Erlangen, June 1996.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A User Guide to HyTech. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 41–71, 1995.

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[Hib] Hibernate. An O/R-Mapper. http://www.hibernate.org.

[Hol87a] Gerard J. Holzmann. Automated Protocol Validation in Argos: Assertion Proving and Scatter Searching. *Software Engineering*, 13(6):683–696, 1987.

[Hol87b] G.J. Holzmann. On Limits and Possibilities of Automated Protocol Analysis. In H. Rudin and C. West, editors, *Proc. 6th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Zurich, Sw., June 1987.

[Hol88] Gerard J. Holzmann. An Improved Protocol Reachability Analysis Technique. *Software - Practice and Experience*, 18(2):137–161, 1988.

[Hol90] G. Holzmann. Algorithms for Automated Protocol Validation, 1990.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HP89] G.J. Holzmann and J. Patti. Validating SDL Specifications: An Experiment. In C. Vissers and E. Brinksma, editors, *Proceedings of the 9th International Conference on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 317–326, Twente, Netherlands, June 1989.

[HS96] T. Hadlich and T. Szczepanski. The ODE System - An SDL Based Approach to Hardware-Software Co-Design. *Embedded Microprocessor Systems*, pages 269–281, 1996. IOS Press, Amsterdam, Netherlands.

[IEE93] IEEE. VHSIC Hardware Design Language. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, USA, 1993.

[IEE95] IEEE. IEEE Standard Hardware Design Language Based on the Verilog Hardware Description Language. IEEE 1364. Institution of Electrical and Electronic Engineers Press, New York, USA, 1995.

[ISO88] ISO - International Organization for Standardization - Information Proceedings Systems - Open Systems Interconnection. ISO/IEC. LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, 1988.

[ITU93a] ITU - International Telecommunication Union. Message Sequence Chart (MSC), ITU-T Recommendation Z.120. Geneva, Switzerland, 1993.

[ITU93b] ITU - International Telecommunication Union. Specification and Description Language (SDL), ITU-T Recommendation Z.100. Geneva, Switzerland, 1993.

[ITU94a] ITU - International Telecommunication Union. Specification and Description Language (SDL), ITU-T Recommendation Z.100 Annex F.2 - SDL Formal Definition: Dynamic Semantics. Geneva, Switzerland, 1994.

[ITU94b] ITU - International Telecommunication Union. Specification and Description Language (SDL), ITU-T Recommendation Z.100 Annex F.2 - SDL Formal Definition: Static Semantics. Geneva, Switzerland, 1994.

[ITU01] ITU - International Telecommunication Union. Tree and Tabular Combined Notation (TTCN), ITU-T Recommendation Z.140. Also: International Electrotechnical Commission [IEC] 9646-3, 2001.

[JLS96] H. Jensen, K. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL, 1996.

[Joc02] M. Jochim. Pessimistic Fault Injection for Automatically generated Virtual Duplex Systems. In *Proceedings of ARCS-02*, pages 25–29, 2002.

[Jon99] T. Jones. A Tool-suite for Simulating, Composing and Editing Timed Automata (LUSCETA: User's Manual Release 1.0). Technical Report MPG-99-24, Lancaster University, Lancaster, LA1 4YR, 1999.

[JPP+97] Lalita Jategaonkar Jagadeesan, Adam A. Porter, Carlos Puchol, J. Christopher Ramming, and Lawrence G. Votta. Specification-based Testing of Reactive Software: Tools and Experiments (Experience Report). In *International Conference on Software Engineering*, pages 525–535, 1997.

[JRV+97] A.A. Jerraya, M. Romdhani, C.A. Valderrama, P. LeMarrec, F. Hessel, G.F. Marchioro, and J.-M. Daveau. Languages for System-Level Specification and Design. *Hardware/Software Co-Design: Principles and Practice*, pages 235–262, 1997. Kluwer Academic Publishers, London, UK.

[Kes02] P. Kessler. Implicit Fault Modeling with the Specification Language SDL. In *Proceedings of ARCS'02, International Conference on Architecture of Computing Systems*, pages 81–90, 2002.

[KG93] Hermann Kopetz and Günter Grünsteidl. TTP – A Time-triggered Protocol for Fault-tolerant Real-time Systems. In *Proceedings 23rd International Symposium on Fault-Tolerant Computing*, pages 524–532, 1993.

[KG99] C. Kern and M.R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, April 1999.

[Kur97] R.P. Kurshan. Formal Verification in a Commercial Setting. In *Proceedings of the Design Automation Conference*, pages 258–262, Anaheim, CA, USA, June 1997.

[KW91] Thomas Kropf and Hans-Joachim Wunderlich. A Common Approach to Test Generation and Hardware Verification Based on Temporal Logic. In *International Test Conference*, pages 57–66, 1991.

[Laf03] Alberto Lluch Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, 2003.

[Lap92] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 6 of *Dependable Computing and Fault-Tolerant Systems. Prepared by IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance*. Springer-Verlag, 1992.

[LBBI96] V. Levin, E. Bounimova, O. Basbugoglu, and K. Inan. A Verifiable Software/Hardware Co-Design Using SDL and COSPAN. In Z. Brezocnik and T. Kapus, editors, *Proceedings of COST 247 International Workshop on Applied Formal Methods*, pages 6–16, University of Maribor, Slovenia, June 1996.

[LH00] J. Jenny Li and J. Robert Horgan. A Tool Suite for Diagnosis and Testing of Software Design Specifications. In *Proceedings of DSN'00*, pages 295–304, 2000.

[LH02] K. Loer and M.D. Harrison. Towards Usable and Relevant Model Checking Techniques for the Analysis of Dependable Interactive Systems. In W. Emmerich and D. Wile, editors, *Proceedings 17th International Conference on Automated Software Engineering*, pages 223–226. IEEE Computer Society, September 2002.

[LH04] Karsten Loer and Michael Harrison. Integrating Model Checking with the Industrial Design of Interactive Systems, 2004.

[LLEL01] Alberto Lluch-Lafuente, Stefan Edelkamp, and Stefan Leue. Partial Order Reduction in Directed Model Checking. Technical Report 19, Albert-Ludwigs-Universität Freiburg im Breisgau, 2001.

[LPY97a] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL - Validation and Verification of Real Time Systems - Status & Developments, 1997.

[LPY97b] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LSP80] L. Lamport, R. Shostak, and M. Pease. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PB03] P.J. Pingree and E.G. Benowitz. Experiences in Integrating Auto-Translated State-Chart Designs for Model Checking. In *Proceedings of the DSN'04 (Supplemental)*, pages W72–W74, San Francicso, CA USA, 2003. IEEE Press.

[Pol95] S. Poledna. Fault Tolerance in Safety Critical Automotive Applications: Cost of Agreement as a Limiting Factor. In *Proceedings of FTCS-25*, page 73, 1995.

[Rau90] Marko Rauhamaa. A Comparative Study of the Methods for Efficient Reachability Analysis. Technical Report A14, Helsiniki University of Technology, 1990.

[Sev93] R.E. Seviora. Real-Time Detection of Failures of Reactive Systems. In Janusz Gorski, editor, *SAFECOMP'93: 12th International Conference on Computer Safety, Reliability and Security*, pages 265–275, Poznan-Kiekrz, Poland, 1993. Springer-Verlag.

[SL97] M. Steppler and M. Lott. SPEET — SDL Performance Evaluation Tool. In Cavalli and Sarma [CS97], pages 53–67.

[SRSP04] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. In *Proceedings of the DSN'04*, pages 189–198, 2004.

[TAML00] Kenneth J. Turner, F. Javier Argul-Marin, and Stephen D. Laing. Concurrent Specification and Timing Analysis of Digital Hardware Using SDL. *Lecture Notes in Computer Science*, 1800:1001–1008, 2000.

[TCL99] K. Turner, G. Csopaki, and S. Laing. Hardware Timing Analysis with SDL, January 1999.

[Tel01] Telelogic AB. *Telelogic Tau 4.2 SDL Suite; Getting Started*. Telelogic, Malmö, Sweden, 2001.

[Tou84] Sam Toueg. Randomized Byzantine Agreements. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of Distributed Computing*, pages 163–178, New York, NY, USA, 1984. ACM Press.

[TTT03] TTTech. TTP - Time Triggered Protocol TTP/C High-Level Specification Document. Protocol Version 1.1., November 2003.

[Wal96] L. Walther. *Die Erreichbarkeitsanalyse zur Validierung komplexer Kommunikationsprotokolle*. PhD thesis, Universität der Bundeswehr, München, 1996.

[Yov97] Sergio Yovine. Kronos: A Verification Tool for Real-Time Systems. (Kronos User's Manual Release 2.2), 1997.

[YTK01] T. Yokogawa, T. Tsuchiya, and T. Kikuno. Automatic Verification of Fault Tolerance Using Model Checking. In *PRDC*, pages 95–102, 2001.

[Zaf77] P. Zafiropulo. A New Approach to Protocol Validation. In *International Conference on Communications*, Chicago, June 1977.

Acronyms

2SP	2-Switch Protocol	109
α	Influence of the input weight on the overall transition weight	70
$\mathcal{A}_j$	Set of all action elements contained in transition tr_j	69
$\mathcal{A}_{UNIQUE}$	Set containing only one instance of each action element in tr_j	69
$action(tr_{j,k})$	The k^{th} action element in transition tr_j	69
$activeTRset(s_i)$	Active transitions, i.e. all transitions that are able to fire in state s_i	57
AE_P	Pure Action Weights	142
AI	Input Weights Extending Action Weights	144
β	Influence of the action weight on the overall transition weight	70
C1, C2	Conditions for SFR-PO analysis	119
C2F	Close-to-Failure	75
$\text{C2F}_{\text{PART}-\text{F}}$	Variant of C2F requiring only minimum model knowledge	78
C2F_{PRED}	Variant of C2F allowing access to the predecessor state	78
DBA1	Deterministic Variant of RBA1	104
$depth(s_i)$	Depth of global state s_i within the state space	62
$df(depth(s_i))$	Depth reduction factor based $depth(s_i)$	62
$\mathcal{E}$	Set of user-defined failure property rules	76
e_i	Failure property rule i	76
$e_i(s_j)$	Evaluation of e_i for global state s_j	76
$e_{AVG}(s_j)$	Combining function of the e_i using the average	76
$e_{MAX}(s_j)$	Combining function of the e_i using the maximum	76
EQ	Action Weights and Input Weights Equally	145
f	User-defined final rule for protocol validation	75
F	Number of faulty components	101
$f_{action}(tr_j)$	Overall action weight calculation function	69
$fired(tr_j)$	Number of times tr_j fired on the current path	67
FX	FlexRay Protocol	116
HC	Combined H-RAFT and C2F algorithm	156
H-RAFT	Heuristic Reachability Analysis of Fault-Tolerant Systems	55
IA	Action Weights Extending Input Weights	145
IE	Input Elements	133
IE_{CENTER}	Optimum selected as center for neighborhood investigation	134
$IE_{OPTIMUM}$	Optimum weight combination	133
$input(tr_j)$	Input element of transition tr_j	64
$nDec$	Decrease factor for spontaneous transitions	67
PP	Pendulum Protocol	97
RAFT (Tool)	Reachability Analysis of Fault-Tolerant Systems	83

RBA1	Randomized Byzantine Agreement Protocol	103
s_i	Global state i	57
s_{next}	The state to be chosen next	57
$\mathcal{S}_{next}$	Set of selectable global states	58
SCO	Signal-Consumption-Only transition	67
$\mathcal{SCO}$	The set containing all Signal-Consumption-Only transitions	67
SDL	Specification and Description Language	11
SFR-PO	Single Fault Region Partial Ordering	37
SM	Signed Messages Protocol	101
$\mathcal{ST}$	Set containing all special transitions	73
$stateSpace_{curr}$	Already explored state space	57
$stateSpace_{curr,act}$	Set of active transitions in the already explored state space	57
SUM	Single Action Weights and Single Input Weights	146
$\mathcal{T}_{next}$	Set of selectable transitions	63
tr_j	Transition j	59
tr_{next}	The transition to be fired next	57
VETO	VETO Protocol	107
w_i	Static weight values	132
$wAction(action)$	Static weight of the action element	69
$wAction(tr_j)$	Total action weight of transition tr_j	69
$width$	Factor for width restriction of the reachability graph	124
$wInput(input(tr_j))$	Static weight of input element $input(tr_j)$	65
$wSpecial(tr_j)$	Weight for special transition tr_j	73
$wState(s_i)$	Weight of global state s_i	57
$wTrans(tr_j)$	Weight of tr_j	59